1

DESIGN OF AN ALGORITHM TO TRANSLATE NESTE RELATIONAL ALGEBRA QUERIES TO GENESIS TRACE MANAGER COMMANDS

THESIS

Alan Frank Hartman
Captain, USAF

AFIT/GCS/ENG/87D-13

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

PII Redacted

1

AFIT/GCS/ENG/87D-13

DESIGN OF AN ALGORITHM TO TRANSLATE NESTED RELATIONAL ALGEBRA QUERIES TO GENESIS TRACE MANAGER COMMANDS

THESIS

Alan Frank Hartman
Captain, USAF

AFIT/GCS/ENG/87D-13

AFIT/GCS/ENG/87D-13

DESIGN OF AN ALGORITHM TO TRANSLATE NESTED RELATIONAL ALGEBRA QUERIES TO GENESIS TRACE MANAGER COMMANDS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Information Systems)

Alan Frank Hartman, M.S.

Captain, USAF

December, 1987

Preface

The goal of this thesis was to develop an interface between nested relational algebra queries and the GENESIS Trace Manager, which is part of the GENESIS database management system being developed at the University of Texas at Austin, Texas.

Although this thesis does not include an implementation of the interface, it does present a group of algorithms that can be used to implement the interface. The algorithm consists of two phases, in the first phase the nested relational algebra query is converted into an intermediate data structure and in the second phase the intermediate data structure is used to generate the GENESIS Trace Manager commands.

I am deeply indebted to my thesis advisor, Captain Mark Roth, for his invaluable assistance during the development of this thesis. I also wish to thank the other members of my committee, Captain Wade Shaw and Dr. Thomas Hartrum, for their assistance. In addition, I wish to thank Jim Barnett, of the University of Texas at Austin, Texas for his assistance with the GENESIS Trace Manager.

Alan Frank Hartman

Table of Contents

List of Figures

Figure Page

Figure Page

AFIT/GCS/ENG/87D-13

Abstract

This thesis describes an algorithm to convert nested relational algebra queries into GENESIS Trace Manager commands. Nested relational algebra is an extension to traditional relational algebra to include multivalued (i.e. nested) attributes. The GENESIS Trace Manager is part of the GENESIS database management system being developed at the University of Texas at Austin, Texas. The GENESIS Trace Manager is used to manipulate fields in a record that has been read into a buffer in memory.

The algorithm consists of two phases. The first phase of the algorithm is the development of an intermediate data structure to represent the various constructs of the nested relational algebra query. The second phase of the algorithm is the convefsion of the intermediate data structure into GENESIS Trace Manager commands. This phase consists of dividing the translation into a number *of sub-tasks and providing an algorithm to perform each of these sub-tasks.*

The GENESIS Trace Manager is limited to working with fields in a record located in a buffer in primary memory. It does not include facilities for reading records from a database into memory, writing records from memory to a database, or presenting the user with a formated output of the result of the query. Because the GENESIS Trace Manager does not include these facilities, the algorithm does not produce GENESIS commands to perform these functions. [illegible]

DESIGN OF AN ALGORITHM TO TRANSLATE NESTED RELATIONAL ALGEBRA QUERIES TO GENESIS TRACE MANAGER COMMANDS

I. Introduction

1.1 Current Research in Relational Databases

Classical relational database models have been very useful for working with many database applications. Most current relational database models assume that relations are in first normal form (1NF) where all attributes must be atomic (single-valued). This assumption has made it difficult to implement databases for applications such as office forms, computer-aided design, and statistical databases. Current research [17,22] indicates that these databases may be better represented by relational models that allow an attribute to contain multiple values or another relation. These models are referred to as nested relational models or non-1NF models.

Developing a database for these types of applications usually involves adding extensions to an existing database or developing an application specific database. The first approach often leads to an inefficient implementation and the second approach is often time consuming and expensive. In order to overcome these problems, the GENESIS database management system (DBMS) is currently being developed at the University of Texas at Austin [2].

GENESIS is a reconfigurable DBMS which supports nested relations. The goal of the GENESIS project is to provide an environment which supports the efficient development of databases. A GENESIS database is built from software modules which are maintained in a software library. The GENESIS software library facilitates database development by allowing software modules to be reused to develop new databases and by allowing a database to be easily reconfigured by selecting different software modules from the library.

1.2 Thesis Goal

One of the key factors in developing a database is designing a user interface to the database. The user interface includes a query language which allows the user to retrieve information from the database. For a user to be able to access a GENESIS database containing nested relations, there must be an interface between GENESIS and a query language which supports nested relations. One of the most popular query languages is the Structured Query Language (SQL) [4]. Although the basic SQL language does not support nested relations, the SQL language has been extended to SQL non-1NF (SQL/NF) [25] to include nested relations.

Currently there is no complete interface between SQL/NF and GENESIS. However, parts of this interface have been developed. Ramakrishnan [24] has developed an SQL/NF translator which converts SQL/NF expressions into nested relational algebra expressions. Smith [27] has developed a GENESIS Trace Manager which can be used to access fields within records. Currently there is no interface between the nested relational algebra which is output by the SQL/NF translator and the GENESIS Trace Manager.

The original goal of this thesis was to design and implement an algorithm to translate nested relational algebra queries to GENESIS Trace Manager commands. During the development of this thesis the scope was limited to the design and evaluation of an algorithm, and does not include an implementation of the algorithm. This change in scope was due in part to time constraints and in part to the fact that other sections of the GENESIS DBMS were not finished at the time of this thesis. The sections of the GENSIS DBMS that create, access and maintain the database were not available for use in this thesis. The implementing and testing of the algorithm would have required the design and implementation of a DBMS and report generator. These tasks were not within the scope of this thesis.

1.3 Thesis Outline

Because the concept of nested relations is central to this thesis, Chapter 2 provides an introduction to nested relations, including descriptions of the classical relational database model, normal forms of relational models, the advantages of nested relational models, applications of nested relations, and nested relational models. Chapter 3 describes SQL/NF, nested relational algebra and the SQL/NF translator which converts SQL/NF queries into nested relational algebra queries. Chapter 4 describes the GENESIS Trace Manager which includes the GENESIS Data Definition Language and the GENESIS Trace Manager. Chapter 5 describes the algorithm, design decisions made during development of the algorithm, validation of the algorithm, and analysis of the performance of the algorithm. Chapter 6 presents conclusions and recommendations for further research.

II. The Relational Model and Nested Relations

A database model is a way of looking at data at the logical level. A number of database models have been proposed including the entity-relationship model [5], the network model [28] and the hierarchical model [29]. The relational model, which was introduced in 1970, is a relatively new model. Since 1970, the relational model has been widely discussed in the literature and has developed into the most popular database model. Recently the relational model has been extended to include nested relations. This chapter provides a description of the relational database model, relational model design, normal forms, applications of nested relations, and nested relational models.

2.1 Relational Database Model

The relational database model was originally defined in 1970 by Codd [6]. Since 1970, a large body of work has developed around the relational model. In this section, we will provide informal definitions of some key terms and describe some of the characteristics of the relational model. The discussion in this section is based on Codd [9] and Korth and Silberschatz [19]. A more formal mathematical treatment of the relational model can be found in Maier [20] or Yang [31]. We will begin our discussion of the relational model by providing the following informal definitions:

- A relation is a set of data represented as a table.
- An attribute corresponds to an object or characteristic in the real world and is represented by a column in the table.
- A tuple corresponds to a relationship between attributes and is represented by a row in the table.

In a relational database model, information is represented in the form of one or more tables, such as the one shown in Figure 1. The name "relation" refers to the fact that each row in the table represents a relationship between attributes. The table in Figure 1 is an example of a relation

NAME	AGE	ADDRESS	CITY	STATE
John Smith	43	123 Main St.	New York	New York
Mary Jones	27	456 Oak St.	Chicago	Illinois
Sam Green	51	789 Elm St.	Dallas	Texas

Figure 1. PERSON Relation

called PERSON, where each column represents an attribute and each row represents a tuple. In this relation each person has five attributes: NAME, AGE, ADDRESS, CITY, and STATE. Each tuple represents the relationship between these five attributes for a given person.

Any data item (such as John Smith's age) in a relation can be referenced by the following set of data:

1. The relation name (PERSON).
2. One or more key fields that specify the tuple (NAME = "John Smith").
3. Attribute name (AGE).

This type of reference, called associative addressing, gives the relational model the following desirable features.

- The relational model allows users to deal with the database at the logical level, without being concerned with the low-level physical implementation of the database.
- The relational model facilitates communication between users and programmers by allowing them to reference data in the same logical manner.
- The relational model provides a basis for the development of a high level language which is independent of the underlying database structure.

In addition to the desirable features described above, a good relational model should provide an efficient representation of data. For example, an efficient model should avoid duplication of data, and should allow data to be easily updated. The efficiency of a relational model depends on its design. In the next section, we provide an informal description of relational model design.

2.2 Relational Model Design and Normal Forms

One of the goals in designing a relational model is to provide an efficient representation of data. In this section, we will provide an informal introduction to relational model design theory based on Maier [20] and Date [11]. Additional sources of information on relational model design include Date [10] and Ullman [30].

The desire to design efficient relational models has led to the development of normalization theory. Normalization theory is based on a number of normal forms. A normal form is a set of criteria designed to prevent a relational model from having certain undesirable properties. A relation is said to be in a specific normal form if it satisfies the criteria of that normal form. A number of normal forms have been defined. First normal form (1NF), second normal form (2NF), third normal form (3NF), and Boyce/Codd normal form (BCNF) were defined by Codd [7,8]. Fourth normal form (4NF) and fifth normal form (5NF) were defined by Fagin [12,13].

A relational model is said to be in 1NF if all the attributes in the relation are atomic. By atomic we mean that an attribute can contain only a single, non-decomposable value. An attribute that is not atomic is nested. A nested attribute may contain a group of values or another relation.

As an example, consider a relation containing the attributes NAME, and SEX. Figure 2 shows a 1NF relation for this data and Figure 3 shows a nested (or non-1NF) relation for this data. Figure 3 is not in 1NF because the attribute NAME contains nested values.

One of the reasons that traditional databases have used 1NF relations is that they avoid some problems that can occur during the update of a database. For example, assume that the data for Jean's SEX was incorrect and should be changed from male to female. This update can be done

NAME	SEX
Sam	male
Jean	male
Bob	male
Pam	female
Jane	female

Figure 2. 1NF Relation

NAME	SEX
Sam Jean Bob	male
Pam Jane	female

Figure 3. Nested Relation

easily in the 1NF relation in Figure 2 by changing the value for Jean's SEX from male to female, as shown in Figure 4.

The update to the nested relation in Figure 3 is not as straight forward. If the entry for Jean's SEX is changed from male to female, as shown in Figure 5 then the SEX data for John and Bob is incorrect since it indicates that John and Bob are female.

The purpose in presenting the above example was to show an advantage of 1NF relations with respect to nested relations. It should not be concluded from this example that 1NF relations are always easier to update than nested relations. There are other situations in which nested relations

NAME	SEX
Sam	male
Jean	female
Bob	male
Pam	female
Jane	female

Figure 4. Updated 1NF Relation

NAME	SEX
Sam	female
Jean	
Bob	
Pam	female
Jane	

Figure 5. Incorrect Updated Nested Relation

are easier to update than 1NF relations. Examples of these situations will be presented later in this chapter.

In normalization theory, each of the normal forms after 1NF adds further requirements. Figure 6, from Date [11, page 363], shows a graphical representation of this concept. For example, Figure 6 shows that 5NF requires that a relation also be in 1NF, 2NF, 3NF, BCNF, and 4NF. All of the normal forms listed require that relations be in 1NF. It should be noted that we are referring here to traditional descriptions of normal forms. We will show later that it is possible to extend normal forms such as 3NF to deal with nested relations.

All the relational models we have shown so far have consisted of a single relation. Dividing a relational model into more than one relation may produce a more efficient design. A relational model can be put into 1NF without dividing the model into multiple relations. However, a relational model may have to be divided into multiple relations to satisfy the criteria of 2NF, 3NF, BCNF, 4NF, or 5NF. Formal definitions for 2NF, 3NF, BCNF, 4NF, and 5NF are provided in Maier [20] and Yang [31]. An intuitive description of 1NF, 2NF, 3NF, 4NF, and 5NF can be found in Kent [18]. We will not provide formal definitions here, but will provide an example to show how dividing a relational model into multiple relations can produce a more efficient design.

As an example, consider a relational model for students attending classes, where the attributes are STUDENT, COURSE, course TEACHER, course LOCATION, and course TIME. In this example, we will assume that each course has only one TEACHER, one LOCATION, and one

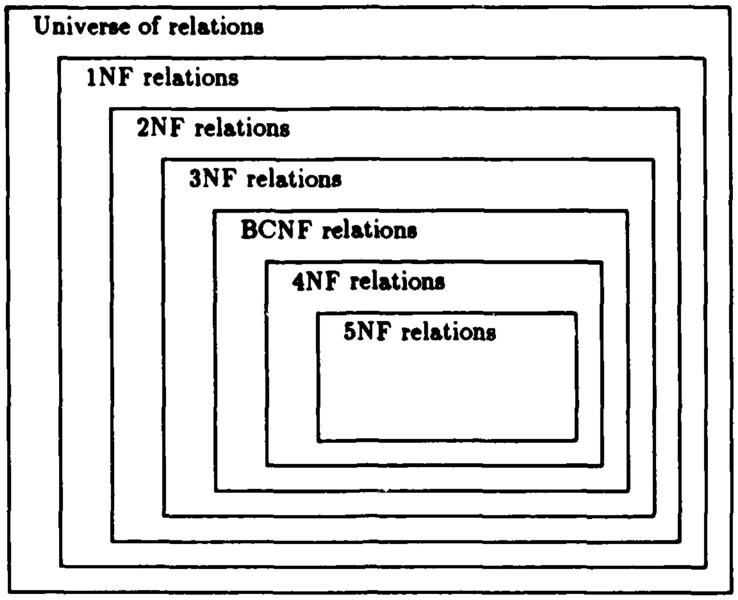


Figure 6. Hierarchy of Normal Forms [11, page 363]

TIME. It is possible to include all of these attributes in a single relation, as shown in Figure 7. However, this representation of the data is not efficient. One problem with this representation is that the information on TEACHER, LOCATION, and TIME is duplicated for each student. Elimination of this duplication would make the model more efficient. Another problem with this model is that it is difficult to modify. As an example, assume that the LOCATION and TIME for MATH 100 changed. In order to update the database, LOCATION and TIME must be modified for each occurrence of MATH 100. If the data on LOCATION and TIME for each COURSE only occurred once in the database, then updating the database would be more efficient because only one occurrence of LOCATION and TIME would need to be modified.

The STUDENT-COURSE data can be represented more efficiently by dividing the data into two relations as shown in Figure 8. The relational model in Figure 8 is more efficient because it eliminates the redundancy of TEACHER, LOCATION, and TIME that occurs in Figure 7.

STUDENT	COURSE	TEACHER	LOCATION	TIME
John Smith	Math 100	Russel	Room 111	9:00 AM
Mary Green	Math 100	Russel	Room 111	9:00 AM
Jim Jones	Math 100	Russel	Room 111	9:00 AM
Janice Johnson	Chem 200	Pauling	Room 222	10:00 AM
Bob White	Chem 200	Pauling	Room 222	10:00 AM
Pam Adams	Chem 200	Pauling	Room 222	10:00 AM

Figure 7. STUDENT-COURSE Data Represented with One Relation.

STUDENT	COURSE
John Smith	Math 100
Mary Green	Math 100
Jim Jones	Math 100
Janice Johnson	Chem 200
Bob White	Chem 200
Pam Adams	Chem 200

COURSE	TEACHER	LOCATION	TIME
Math 100	Russel	Room 111	9:00 AM
Chem 200	Pauling	Room 222	10:00 AM

Figure 8. STUDENT-COURSE Data Represented with Two Relations.

Also, the relational model in Figure 8 is easier to update because the information on TEACHER, LOCATION, and TIME only occurs once for each COURSE.

The purpose in presenting the example above was to show the advantage of dividing a relation into two relations. However, it should be noted that dividing a relation into two relations also has a disadvantage. The process of retrieving all the information about a student may be less efficient because this information is divided between two relations.

Normalization theory provides a formal treatment of criteria for subdividing a set of attributes into multiple relations. The normal forms after 1NF are sets of criteria designed to provide an efficient means of performing this subdivision.

PARENT	CHILD
John Smith Mary Smith	Paul Elizabeth Peter
Sam Jones Cathy Jones	Thomas Jane

Figure 9. PARENT-CHILD as a Nested Relation.

In this section we have assumed that all relations were in 1NF. In the next section we will describe some advantages of nested relations and give examples of applications where nested relations can be used.

2.3 Applications of Nested Relations.

Traditional relational models have used 1NF relations because they are conceptually simpler and because they are adequate for many traditional business applications. Although it may be more difficult to implement nested relations, there are potential advantages to nested relations. Nested relations can eliminate redundancy and more accurately reflect real world objects and attributes.

Many real world situations deal with nested attributes. For example, a parent may have more than one child. A nested relational model could represent a PARENT-CHILD relation as shown in Figure 9. In this representation, PARENT and CHILD are nested attributes. In a 1NF relational model the information would be represented as shown in Figure 10. The nested relation in Figure 9 is more efficient than the 1NF relation in Figure 10 because there is no duplicated data. Another advantage of the nested relation is that it more accurately reflects the relationship between parents and children.

The current interest in nested relational models is due to the desire to develop an efficient model which supports nested relations. The current literature includes numerous articles dealing with application of nested relations to areas such as form flow design [17] and statistical databases

PARENT	CHILD
John Smith	Paul
John Smith	Elizabeth
John Smith	Peter
Mary Smith	Paul
Mary Smith	Elizabeth
Mary Smith	Peter
Sam Jones	Thomas
Sam Jones	Jane
Cathy Jones	Thomas
Cathy Jones	Jane

Figure 10. PARENT-CHILD as a 1NF Relation.

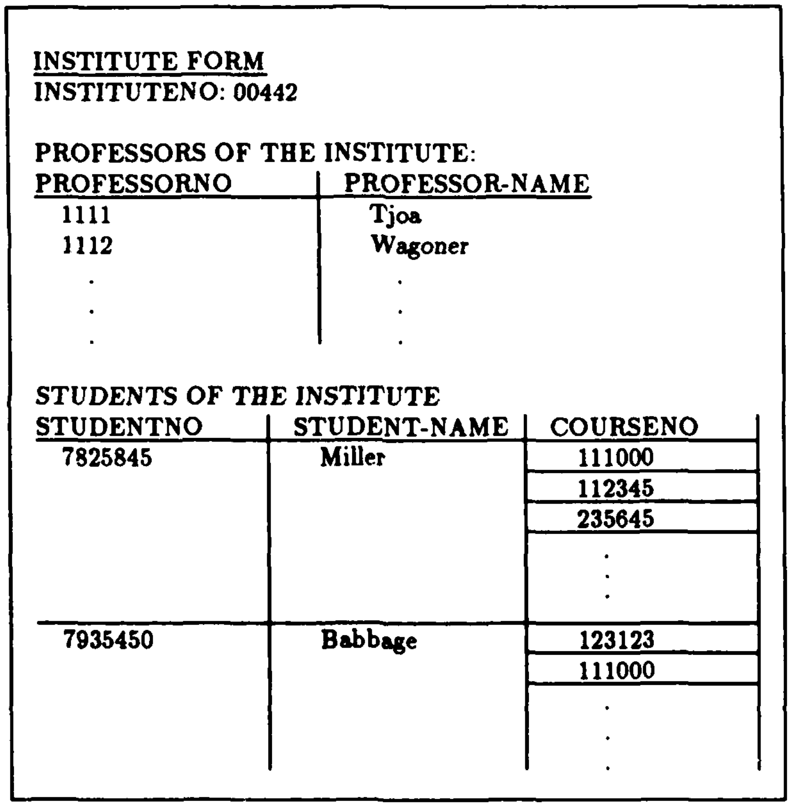
[22,23]. The following sections describe some of these applications that can benefit from using nested relational models.

2.3.1 Form Flow Design. Kappel et al. [17] have shown that a nested relational model is advantageous in designing form flow systems. Kappel et al. use the term non-first-normal-form (NF2) to refer to nested relations. In order to be consistent with our earlier terminology, we will continue to use the term nested.

A form can be viewed as a relation. If the form contains nested attributes, then it can be viewed as a nested relation. Figure 11, taken from [17, page 236], shows a sample institute form. Figure 12, from [17, page 240], shows a nested relation which includes the data from the form in Figure 11 and additional sample data.

Kappel et al. point out the advantages of nested relations by comparing them to flat (i.e. unnested) relations as described by Schmid and Swenson [26]. The nested relation is preferable to a flat relation because all the attributes associated with a given object can be represented in a single relation, and there is less repetition of data.

The nested relational model described above has been used in the development of a prototype form flow system. The prototype is called Computerunterstutztes Buroinformationsssytem (CBIS)



INSTITUTE FORM
INSTITUTENO: 00442

PROFESSORS OF THE INSTITUTE:

PROFESSORNO	PROFESSOR-NAME
1111	Tjoa
1112	Wagoner
. . .	. . .

STUDENTS OF THE INSTITUTE

STUDENTNO	STUDENT-NAME	COURSENO
7825845	Miller	111000
		112345
		235645
		. . .
7935450	Babbage	123123
		111000
		. . .

Figure 11. Institute Form [17, page 236]

		PROFESSOR		STUDENT		
Institute_No	Institute_Name	Professor_No	Professor_Name	Student_No	Student_Name	Course
00442	Comp.Science	1111	Tjoa	7825645	Miller	111000
		1112	Wagner			112345
						235645
				7935450	Babbage	123123
						111000
00443	Mathematics	1122	Gauss	7825888	Bit	111111
		1144	Cantor			111222
						111333
				7944444	Byte	111333
						111555

Figure 12. Institute Relation [17, page 240]

and was implemented at the University of Vienna. CBIS has been shown to be an effective system for dealing with university administration data.

2.3.2 Statistical Databases. Ozsoyoglu and Ozsoyoglu [22 23] have shown that it is beneficial to apply nested relations to statistical databases (SDBs). SDBs are used to support a variety of statistical analysis such as sum and average. Sum and average are examples of aggregate functions which analyze a group of values to produce a single valued output. One of the essential constructs of SDBs is the summary table, which is used to maintain and analyze summary data. A sample summary table from [23] is shown in Figure 13.

Most current SDBs have the ability to generate summary tables, but do not store or manipulate information as summary tables. Instead, they store information as atomic values and create summary tables as output to be viewed by the user. It is useful to apply nested relations to summary tables, because summary tables deal with nested attributes. Ozsoyoglu and Ozsoyoglu [23] propose applying relations with set-valued attributes (i.e. nested relations) to SDBs to provide

Ohio	Ashtabula	Female	63.2
		Male	77.2
	Cuyahoga	Female	81.5
		Male	56.2
	Medina	Female	61.8
		Male	62.4
	Ashtabula		68.9
	Cuyahoga		60.4
	Medina		62.0
Pennsylvania	Allegheny	Female	79.3
		Male	70.2
	Susquehanna	Female	66.0
		Male	70.0
	Allegheny		75.4
	Susquehanna		68.1

Figure 13. Summary Table, Average-House-Prices in Thousands of Dollars.

the ability to manipulate information as summary tables. The concepts presented in [23] are being used to develop a SDB named System for Statistical Databases.

The examples described above for forms flow design and statistical databases have pointed out some of the advantages of nested relations over 1NF relations. Because of the advantages of nested relations, various authors have extended the classical relational model to include nested relations. Some of these extensions are described in the next section.

2.4 Nested Relational Models

A number of articles have appeared in the literature which extend the relational model to include nested relations. Makinouchi [21] showed that traditional normal forms such as 3NF and 4NF could be extended to include sets of sets. He demonstrated that the mathematical interpretation of 3NF and 4NF do not necessarily imply 1NF. He introduced extensions to Codd's definition of functional dependency (FD) and Fagin's definition of multivalued dependency (MVD) so that FD and MVD can be applied to nested relations. He used these extended definitions of FD and MVD to define extensions to the traditional definitions of 3NF and 4NF to deal with nested relations.

Jaeschke and Schek [16] provided an extension to the relational model to include power set type relations. They also extended relational algebra, defined NEST and UNNEST operators, and described the properties of these operators. The NEST operator is used to convert a 1NF relation into a nested relation, or to convert a nested relation into a more deeply nested relation. The UNNEST operator is used to convert a nested relation into a less nested relation. If a nested relation is nested only one level deep, then the UNNEST operator converts it into a 1NF relation.

Fischer and Thomas [14] extended the concept of NEST and UNNEST as presented by Jaeschke and Schek [16]. They described the application of the NEST and UNNEST operators, the interaction of relational operators with NEST and UNNEST, and the relationship of functional dependencies with NEST and UNNEST.

Roth [25] defined a normal form called partitioned normal form (PNF). For a relation to be in PNF there must be a series of nest operations than can reverse any series of valid unnest operations. Since nest and unnest operators are central to nested relations it is desirable for a normal form to be closed under these operations. Roth provides proofs that PNF relations are closed for all unnest operations and for a certain class of nest operations. Roth also extends the relational algebra to include PNF operators for union, intersection, difference, Cartesian product, select, natural join, and projection.

Ozsoyoglu and Yuan [22] defined a normal form for nested relations called nested normal form (NNF). For a relation to be in NNF, the relation must be organized as a normal scheme tree. In a normal scheme tree the vertices are pairwise disjoint sets of attributes and the edges correspond to MVDs. Ozsoyoglu and Yuan [22, page 113] provide an example of a nested relation (Figure 14) and its normal scheme tree (Figure 15). A nested field is indicated by the * symbol.

In the scheme tree in Figure 15, the edges represent the following MVDs:

COURSE	(TEXT)*	(SECTION	(DAY)*	(GRADER)*)*
c1	Design Analysis	s1	Mon Wed	John Mary
		s2	Tue Thur	Joe Sue
c2	Data Structure Database	s1	Mon Wed Fri	Sally

Figure 14. Nested Course Relation [21, page 113]

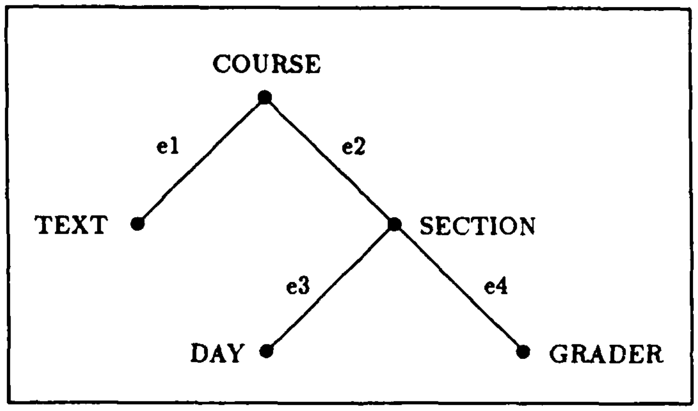


Figure 15. Normal Scheme Tree [21, page 113]

- e1 = COURSE $\twoheadrightarrow$ TEXT
- e2 = COURSE $\twoheadrightarrow$ SECTION, DAY, GRADER
- e3 = COURSE, SECTION $\twoheadrightarrow$ DAY
- e4 = COURSE, SECTION $\twoheadrightarrow$ GRADER

A NNF decomposition of a relation consists of a set of nested relations each of which is a normal scheme tree. This set of normal scheme trees is referred to as a forest of scheme trees.

This chapter has provided a brief introduction to the relational database model including relational model design, various normal forms for relations, some of the advantages of nested relations, applications of nested relations, and extensions to the relational model to include nested relations. In the next chapter we will describe an application for translating a nested relational query from one query language into a different query language.

III. SQL/NF Translator

The SQL/NF translator is a program which converts SQL/NF queries into nested relational algebra queries. This chapter describes the SQL/NF translator and is organized as follows. Section 3.1 describes the SQL/NF query language, Section 3.2 describes the nested relational algebra query language and Section 3.3 describes the method by which the SQL/NF translator converts SQL/NF queries into nested relational algebra queries.

3.1 SQL/NF

The Structured Query Language (SQL) is a relational database language developed for IBM's System R. SQL is used to obtain (i.e. query) information from a database. SQL was designed for 1NF relations and is inadequate for use with nested relations. The SQL/NF is based on SQL, and extends SQL to deal with nested relations [25]. An SQL query consists of three basic parts:

1. A SELECT clause which lists attributes to be output.
2. A FROM clause which lists the relations to be searched.
3. A WHERE clause which specifies the selection criteria.

As an example, assume a user wants to find Roy Wilson's age from the relation in Figure 16. This information could be obtained with the following SQL query:

```
SELECT Age
FROM Name-Age-SSN
WHERE name = "Wilson, Roy"
```

The output of this query would be 34.

SQL/NF extends SQL by including nest and unnest operators. The nest operator is used to convert a 1NF relation into a nested relation or to convert a nested relation into a more deeply nested relation. The unnest operator is used to convert a nested relation into less nested form. If a

Name	Age	SSN
Douglas Hill	23	123-45-6789
Roy Wilson	34	123-45-6789
Carol Crosby	61	123-45-6789

Figure 16. Name-Age-SSN Relation

emp.name	child.name	child.dob
Mary Taylor	Peter	28/02/79
Bob Harris	Jim	11/12/81
Mary Taylor	Jerry	21/07/81
Mike Owens	Bill	22/09/81
Bob Harris	Pam	09/03/83
Mike Owens	Mary	19/11/83

Figure 17. Employees Relation in 1NF Form

nested relation is nested only one level deep, then the unnest operator converts it into 1NF form. As an example, the nest expression:

NEST Employees ON child.name, child.dob as Children

would convert the relation in Figure 17 into the relation in Figure 18, and the unnest expression:

UNNEST Employees on Children

would convert Figure 18 into Figure 17.

emp.name	Children	
	child.name	child.birthday
Bob Harris	Jim	11/12/81
	Pam	09/03/83
Mary Taylor	Jerry	21/07/81
	Peter	28/02/79
Mike Owens	Bill	22/09/81
	Mary	19/11/83

Figure 18. Employees Relation in Nested Form

Class	Title	Section	
		Number	Instructor
Chem 200	Organic Chemistry	1	Smith
		2	Wilson
		3	Peterson
Math 100	Algebra	1	Jones
		2	Smith
		3	Carlson

Figure 19. Course Relation

Another difference between SQL and SQL/NF is that SQL/NF allows SELECT-FROM-WHERE (SFW) expressions to appear in the SELECT clause and FROM clause. Figure 19 shows a Course relation in which the Section attribute consists of a nested relation containing Number and Instructor.

As an example consider the following query which uses an SFW expression in the SELECT clause.

```
SELECT Class, (SELECT Number
               FROM Section
               WHERE Instructor = "Smith")
FROM Course
```

This query will output all the classes and section numbers taught by an instructor with the name Smith.

3.2 Nested Relational Algebra

This section provides a brief discussion of relational algebra and extensions to relational algebra to include nested relations. For a more detailed description of relational algebra, the reader may refer to Korth and Silberschatz [19], Date [11], or Yang [31].

Relational algebra uses operators to query a database. Relational algebra includes five basic operators:

1. SL - the select operator.

Name	Address	City	State
John Smith	123 Main St.	New York	New York
Mary Jones	456 Oak St.	Chicago	Illinois
Sam Green	789 Elm St.	Dallas	Texas
Cathy Brown	295 Walnut St.	New York	New York
June Wilson	852 State St.	Miami	Florida
Jim Johnson	301 Cherry St.	Chicago	Illinois

Figure 20. Person-Address Relation

2. PJ - the project operator.

3. CP - the cartesian-product operator.

4. UN - the union operator.

5. DF - the difference operator

The SL operator is used to select tuples in a relation that meet a specific criteria. The PJ operator is used to select attributes in a relation. The CP operator takes two relations and forms a new relation that includes all combinations of tuples from the two original relations. The UN operator takes two relations and forms a new relation than includes all the tuples that occur in both relations. The DF operator takes two relations and forms a new relation that includes tuples that occur in the first relation but not the second relation. These operators can be used to query a relation such as the Person-Address relation in Figure 20.

As an example of a relational algebra query, assume a user wants to find the names of all the people who live in Chicago. The user could perform this query by using the select and project operators. The following relational algebra query will list the names of all the people who live in Chicago:

```
PJ Name
SL City = "Chicago"
CP Person-Address
```

The output of this query is Mary Jones and Jim Johnson.

Early versions of relational algebra were designed for 1NF relations and were inadequate for describing nested relations. Jaeschke and Schek [16] extended the relational algebra to include nested relations. They defined NEST and UNNEST operators and described the properties of these operators. Fischer and Thomas [14] described the application of NEST and UNNEST operators, the relationship of NEST and UNNEST with other relational operators, and the relationship of NEST and UNNEST to functional dependencies.

3.3 SQL/NF Translator

This section provides a description of the SQL/NF translator developed by Ramakrishnan [24]. The SQL/NF translator converts SQL/NF expressions into nested relational algebra expressions. The nested relational algebra expressions output by the SQL/NF translator are the input to the algorithm which is developed in Chapter V of this thesis.

The translation of an SQL/NF expression into a nested relational algebra expression occurs in three steps:

1. Query transformation.
2. Pre-processing.
3. Meaning evaluation.

Since the goal of the this thesis is to develop an algorithm to translate the output of the SQL/NF translator into GENESIS record manager commands, we are primarily concerned with the format of the SQL/NF translator output. In this thesis we are not directly concerned with the details of the early stages of the SQL/NF translator. Therefore, the first two steps of the SQL/NF translator will be described only briefly and the third step will be described in more detail.

The query transformation step involves three parts: converting the SQL/NF expression to an intermediate form, name resolution, and role-join processing. Conversion to an intermediate form involves building intermediate data structures to represent the SQL/NF query. The name resolution step involves finding the location of the intermediate data structure that corresponds to a given name in the SQL/NF query. The role-join processing step is not relevant to this thesis and will not be discussed.

The pre-processing step is based on the pre-processing described in Ceri and Gottlob [3]. This step uses set-theory transformations to convert the output from the query transformation step into four basic types of queries: simple, complex, exists, or n-ary. In order to describe these four queries we must introduce the following definitions:

scalar boolean - consists of a boolean expression (AND, OR or NOT) whose operands are scalar predicates or other scalar booleans.

scalar predicate - consists of two plain expressions connected by a comparison operator (<, <= , >, >=, =, <>).

complex predicate - consists of variable and a query connected by a comparison operator.

exists predicate - consists of the EXISTS operator and a query operand.

In a simple query, the where clause may be a scalar boolean or the where clause may be absent. In a complex query, the where clause consists of a complex predicate. In an exists query, the where clause consists of an exists predicate. An n-ary query corresponds to the union, intersection or difference of two or more queries.

The final step in the SQL/NF translator is the meaning evaluation step which is based on the meaning evaluation described by Ceri and Gottlob [3]. The meaning evaluation step converts the query expressions produced by the pre-processing step into nested relational algebra expressions. The translation of each type of query is described below. These translations are taken directly from Ramakrishnan [24].

For a simple query which does not have a where clause the translation is as follows:

Let query Q be the following:

SELECT select

FROM from

The translation is:

PJ [Q.select]
FN [Q.select_fns; ϕ]
CP [Q.from]

Where Q.select_fns represents the aggregate and nest expressions in Q.select.

For a simple query in which the where clause is a scalar boolean:

Let query Q be the following:

SELECT select
FROM from
WHERE where

The translation is:

PJ [Q.select ∪ other(Q)]
FN [Q.select_fns; other(Q)]
SL [Q.where]
CP [Q.from ∪ extrels(Q.where)]

Where extrels(Q.where) = the external relations in Q.where and other(Q) is the attributes in the relations in extrels(Q.where). External relations are relations that do not occur in Q.from but have attributes in Q.where.

For a complex query the translation is as follows:

Let query Q be the following:

SELECT select
FROM from
WHERE left_term comp_op sub_query

The translation is:

PJ [Q.select ∪ other(Q)]
FN [Q.select_fns; other(Q)]
SL [Q.left_term Q.comp_op Q.sub_query.select]
CP [(Q.from - connect(Q)) ∪ meaning(Q.sub_query)]

Where meaning(Q.sub_query) is the meaning evaluation of Q.sub_query, connect(Q) is relations in meaning(Q.sub_query) that are not in Q.sub_query, and other(Q) is attributes occurring in relations in connect(Q) but not in Q.from.

For an exists query the translation is as follows:

Let query Q be the following:

SELECT select
FROM from
WHERE EXISTS sub_query

The translation is:

PJ [Q.select $\cup$ other(Q)]
FN [Q.select_fns; other(Q)]
CP [(Q.from - connect(Q)) $\cup$ meaning(Q.sub_query)]

For an n-ary query the translation is as follows:

Let query Q be the following:
SETOP [$Q_1, Q_2, \ldots, Q_n$]

Where SETOP is UNION, INTERSECTION, or MINUS and Q_i are queries.

The translation is:

tr(SETOP) [$R_1, R_2, \ldots, R_n$]
where
tr(UNION) = UN
tr(INTERSECTION) = IN
tr(MINUS) = DF
R_i = CP [MEANING(Q_i) $\cup$ all_extrels_except_of(Q_i)]

Where all_extrels_except_of(Q_i) is the external relations of all queries except those in Q_i.

This chapter has provided a description of SQL/NF, nested relational algebra, and the SQL/NF translator. The SQL/NF translator produces the nested relational algebra queries which are the input to the algorithm described in Chapter V. The purpose of the algorithm is to generate commands for the GENESIS Record Manager which is described in the next chapter.

IV. GENESIS Record Manager

This chapter describes the GENESIS Record Manager. This description is based on Smith [27] and the GENESIS Record Manger User Manual [1]. The GENESIS Record Manger is composed of:

1. The GENESIS Data Definition Language (DDL)
2. The GENESIS Trace Manager (TM)

The DDL is used to define the database format, the records in the database, and the fields within the records. The Trace Manager is used to access and update the fields in the records that have been read into memory. The record manager does not include facilities to store records in a database or retrieve records from a database. Section 4.1 of this chapter describes the GENESIS DDL including the schema and Field definition table (Fdt). Section 4.2 describes the GENESIS TM including trace variables, trace commands and inverted tree structure.

4.1 GENESIS DDL

The GENESIS DDL is used to describe database formats. These formats are referred to as schemas and include record and field definitions. The DDL compiler uses the schema to build the Fdt. The fields in the records are conceptually represented as an inverted tree structure.

A sample schema from Smith [27, page 19] is shown in Figure 21. Comments in the schema are delineated with /* and */. The DDL includes the following 14 reserved words:

ARRAY	DATABASE	INT	SHORT
BYTE	FILES	OF	TYPES
CHAR	FLOAT	OPTIONS	
DOUBLE	INCLUDE	RGP	

```
/* Sample schema using each syntactic construct */

DATABASE example {

OPTIONS
          primary-key: /* designates key field */
          security; /* designates protected field */

TYPES
          name = ARRAY [20] OF CHAR;
          ADDR = {
                    street            name;
                    city_state        ARRAY [30] OF CHAR;
                    zip               INT;
          }; security;
          addrs = RPG of addr;
          btree = {
                    node_id           INT;
                    left              RPG <1> OF btree;
                    right             RPG <1> OF btree;
          };
          unusual = {
                    threeD_addr       ARRAY [2,2,2] OF addrs;
                    lots_of_ints      RPG (4) OF RPG OF INT;
                    binary_tree       btree;
          };

FILES
          employees {
                    emp_name          name;
                    emp_num           INT primary_key;
                    emp_adr           addr;
                    prev_addresses    addrs;
                    curr_wage         float security;
          };
          strange   unusual;
}.        /* end of database */
```

Figure 21. Sample GENESIS Schema [28, page 19]

There are three sections in a GENESIS schema: OPTIONS, TYPES, and FILES. All schemas are required to have a FILES section, the other two sections are optional.

The OPTIONS section is used to indicate the security of the field or to indicate that a field is a primary key. The option field can also be used to indicate that a file is to be stored in a specific structure such as a B+ tree or a heap.

The TYPES section is used to indicate the data types of fields. GENESIS includes six basic data types, CHAR, BYTE, INT, SHORT, FLOAT, and DOUBLE. In addition to these basic types, GENESIS also includes ARRAY, RPG (repeating group), and structure types. A repeating group may be preallocated, bounded, or unbounded. Preallocated and bounded repeating groups have a maximum number of elements they can contain. The maximum for preallocated repeating groups is indicated by the structure "< maximum >" and for bounded repeating groups by "(maximum)". A bounded repeating group only uses the amount of space required for the current number of elements. A preallocated repeating group sets aside enough space for the maximum number of elements even if there are currently fewer than the maximum number of elements. If the definition of a repeating group does not specify a maximum size, then it is considered to be an unbounded repeating group and may contain any number of elements. An example of each type of repeating group is given below:

RPG<5> of INT is a preallocated RPG with a maximum of 5 elements.

RPG(10) of INT is a bounded RPG with a maximum of 10 elements.

RPG of INT is an unbound RPG with an unlimited number of elements.

The GENESIS DDL compiler uses the schema to build the Fdt which is used at run time to access fields within the records. Figure 22 from Smith [27, page 26] shows the Fdt for the sample schema in Figure 21. The following paragraphs describe the structure of the Fdt.

*** field definition table ***

ind	flags	type	bd1	bd2	bd3	off	len	fstcld	numch	name
0	0	DB	0	0	0	0	0	1	2	example
1	0	FILE	0	0	0	0	0	14	5	employees
2	0	FILE	0	0	0	0	0	11	3	strange
3	0	CHAR	0	0	0	0	1	0	0	sys()
4	0	ARR1	20	0	0	0	20	3	1	street
5	0	ARR1	30	0	0	20	30	3	1	city_state
6	0	INT	0	0	0	50	4	0	0	zip
7	2	ATRC	0	0	0	2	52	4	3	sys2
8	0	INT	0	0	0	2	4	0	0	node_id
9	0	RPG	1	1	0	8	0	19	1	left
10	0	RPG	1	1	0	8	0	20	1	right
11	0	ARR3	2	2	2	6	0	21	1	threeD_addr
12	0	RPG	0	4	0	-2	0	22	1	lots_of_ints
13	0	STRC	0	0	0	-4	0	8	3	binary_tree
14	0	ARR1	20	0	0	2	20	3	1	emp_name
15	1	INT	0	0	0	22	4	0	0	emp_num
16	2	STRC	0	0	0	26	54	4	3	emp_addr
17	0	RPG	0	-1	0	82	0	7	1	prev_addresses
18	2	FLT	0	0	0	-80	4	0	0	curr_wage
19	0	STRC	0	0	0	-4	0	8	3	sys3
20	0	STRC	0	0	0	-4	0	8	3	sys4
21	0	RPG	0	-1	0	-2	0	7	1	sys5
22	0	RPG	0	-1	0	-4	0	23	1	sys6
23	0	INT	0	0	0	-2	4	0	0	sys7

Figure 22. Field definition table (Fdt) of Sample Schema [28, page 26]

Index Column. The index column in the Fdt is an index to the row number of the table. The name of the database always occurs in row 0. The name of the first file always occurs in row 1. All the children of a field will be grouped together in consecutive rows in the table. However, the children of a field do not have to occur directly after the parent row in the table. The children of repeating groups and arrays are listed in the table as a single row which defines the type.

Flags Column. The flags column indicates which options have been set for the element in a row. An option is represented as a binary bit in the flags entry.

Type Column. The type column defines the element type. An entry in this column may be one of the following 13 types:

ARR1 - one dimensional array	DBLE - double precession float
ARR2 - two dimensional array	FILE - file
ARR3 - three dimensional array	FLOAT - floating point
BYTE - byte	INT - integer
CHAR - character	RPG repeating group
FILE - file	STRC - structure
DB - database	

Bounds Columns. The bd1, bd2, and bd3 columns specify the bounds of repeating groups and arrays. For repeating groups, the value in bd1 is the lower bound and the value in bd2 is the upper bound. An unbounded repeating group is indicated by the value -1. For arrays, bd1 specifies the size of the first array dimension, bd2 specifies the size of the second array dimension, and bd3 specifies the size of the third array dimension. A one dimensional array will have a zero in bd2 and bd3.

Offset Column. The offset column specifies the offset of the current field from its parent field. The offset to a specific field in the record is determined at run time by starting at the first field in the record and calculating offsets to successive children until the specified field is found.

Since the offset to a given field may vary from record to record, the Fdt must contain the offset from the parent field and not the offset from the beginning of the record. The offsets in the Fdt are given in terms of bytes. If the offset to the field is not fixed, then the offset column contains a pointer to the offset, which is indicated by a negative number in the offset column.

Length Column. The length column specifies the length in bytes of the current field. If a field is composed of subfields, then the length is the total length of all subfields. If there is meta-data associated with the field, then the length includes the length of the meta-data. The Fdt contains a zero in the length column for variable length fields because the length of a variable length field cannot be determined until run time.

First Child Column. If an element in the Fdt has children, then the entry in the first child column specifies the location of the first child within the Fdt.

Number of Children Column. If an element in the Fdt has children, then the number of children column specifies how many children the element has.

Name Column. The name column gives the name of the element. If an element was not assigned a name in the schema, then the DDL compiler generates a name for the element.

4.2 GENESIS Trace Manager

The GENESIS Trace Manager provides the ability to access and update fields in GENESIS records. The Trace Manager includes traces and trace functions. A trace of a given field within a record represents the path of nodes within the tree that must be traversed to arrive at the given field. A trace function is a GENESIS program that is used to manipulate a GENESIS trace structure or a GENESIS record.

The Trace Manager looks at records as inverted tree structures with the individual fields of the record as the leaves of the tree. This tree structure may be constructed from the Fdt by using

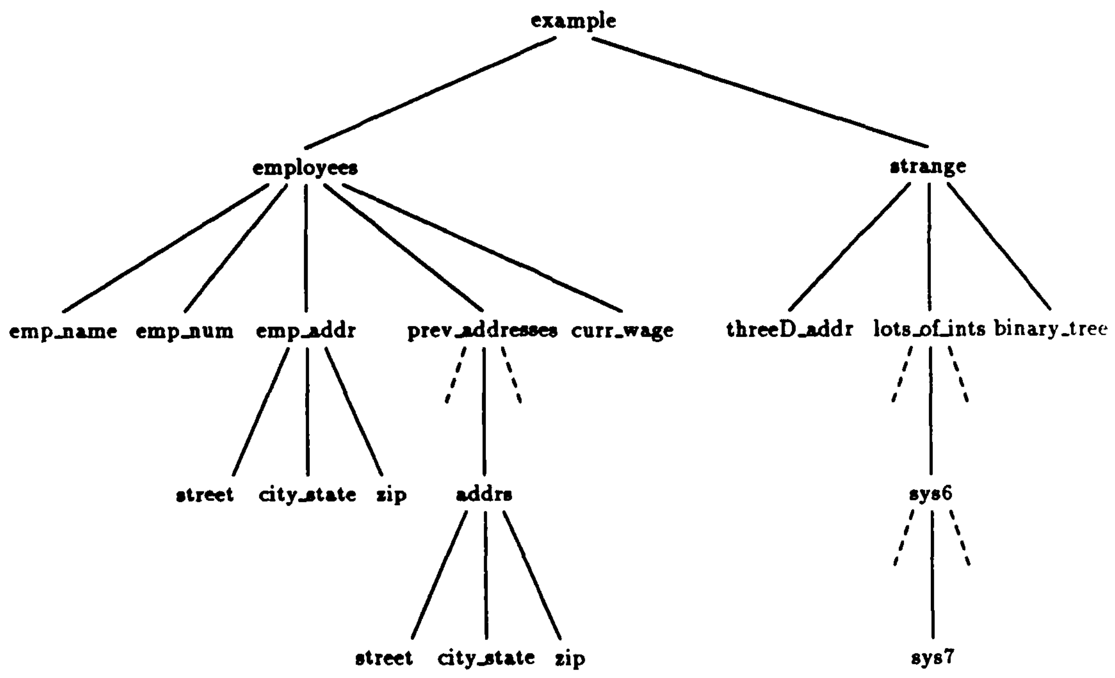


Figure 23. Tree Representation of Fdt

the data in the first child column and the number of children column. Figure 23 gives the tree structure representation of the data in Figure 22. The dashed lines in the tree indicate additional instances of a repeating group. The children of a node are numbered with integers starting at zero. For example, emp_name is child(0) of employees, and emp_num is child(1) of employees.

4.2.1 Trace Variables. A trace variable is a temporary data structure created by the GENESIS Trace Manager to reference fields within a record. Figure 24, taken from Smith [27, page 40], shows an example of a trace variable. The trace contains the path that must be navigated to arrive at the active field. Each node in the path to the active field is represented by an entry in the trace stack. The active field is pointed to by the bottom entry in the stack. A stack entry within the trace consists of three parts:

1. The number representing the child in the current path.
2. The index of the field in the Fdt.
3. The offset of the field from the start of the record.

The GENESIS Record Manager accesses or updates a record via a buffer, which is a temporary storage location in primary memory. A buffer is used to hold a record, or part of a record, that has been read from secondary storage into primary memory. A trace variable must be associated with a buffer in order to access the data in the buffer. When a trace is associated with a buffer, the trace is said to be attached to the buffer. A trace is unattached if it is not associated with a buffer. When a trace is attached to a buffer containing a record, then the trace can be used to read or update a field in the record.

A trace which does not point to an existing field is referred to as a virtually positioned trace. There are two situations that can result in a virtually positioned trace:

1. An unattached trace is virtually positioned because it is not associated with a buffer.
2. If an attached trace points to a valid but nonexistent instance of a repeating group, then it is virtually positioned. As an example, a trace would be virtually positioned if it pointed to the fourth element of an unbounded repeating group which currently contained less than four elements.

In addition to the individual stack entries, a trace variable also contains the following information:

Start. If the trace is not attached to a buffer, then this field contains a zero. If the trace is attached to a buffer, then this field contains the address of the start of the buffer.

Start_level. The start_level field contains the level of the stack that corresponds to the first field in the buffer. For example, if only the third field of a record is read into the buffer, then the start_level will point to the stack entry corresponding to the third field of the record. The

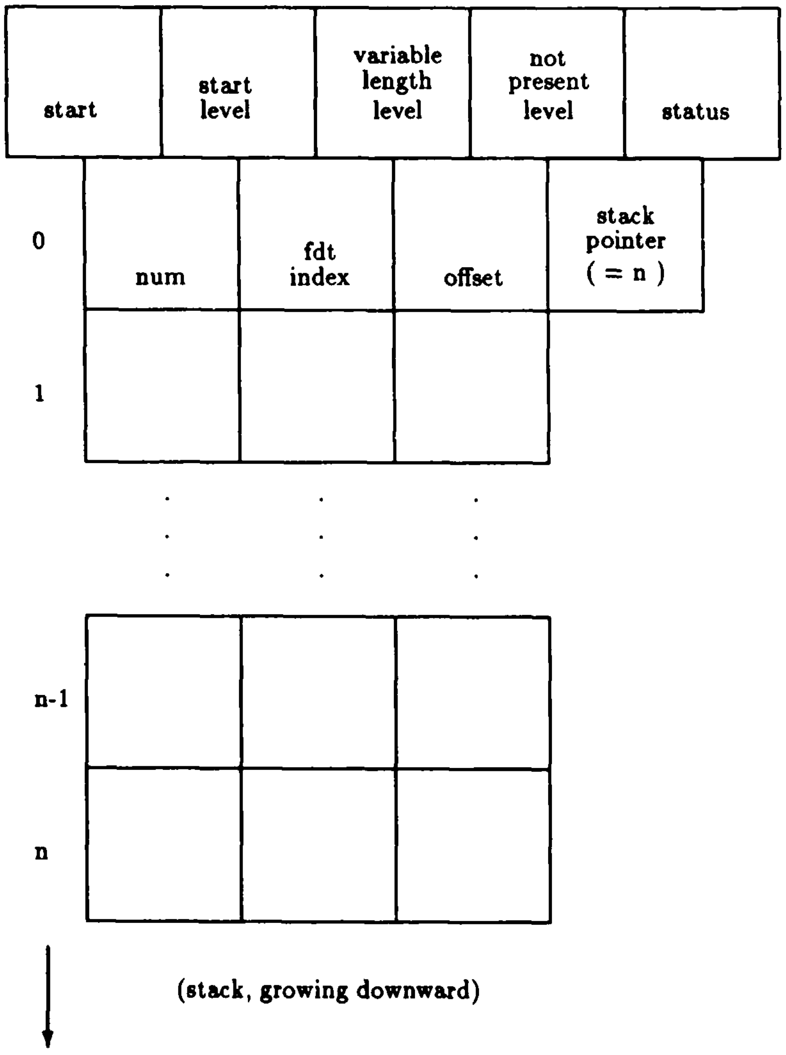


Figure 24. Representation of a Trace Variable [28, page 40]

trace variable will always contain the entire path to the field, even if only part of the record is read into the buffer.

Variable_length_level. This field contains the stack level that corresponds to the field in the trace after the first variable length field. This field will contain zero if there are no variable length fields in the stack or if the only variable length field is the last entry in the stack. The offset of all fields prior to the variable_length_level are fixed. The offset to fields after the variable_length_level will vary from record to record.

Not_present_level. This field contains the stack level of the first virtually positioned field in the trace. The not_present_level will be set to zero if the trace is not virtually positioned.

Status. This field contains the status of the trace. The normal value of this field is OKAY. If an error condition has occurred, then this field contains the appropriate error status. The various error values are listed in the figures in Appendix A.

Stack_pointer. This field points to the bottom entry in the stack.

4.2.2 Trace Functions. Trace functions are GENESIS programs used to manipulate GENESIS traces. There are four types of trace functions: utility functions, navigational functions, information functions, and input/output (I/O) functions. Utility functions are used to create and maintain trace variable structures. Navigational functions are used to position a trace to a parent, child or sibling node within the tree. Information functions are used to obtain data about a field, such as the field length or number of children. I/O functions are used to manipulate fields in a record.

A trace function may return a normal termination condition or an error termination condition in the status field of the trace. A list of trace functions, along with there termination conditions is provided in the figures in Appendix A.

The following paragraphs provide a description of the GENESIS trace functions. In the following descriptions, "t" represents a trace and "buf" represents a buffer.

UTILITY FUNCTIONS

Attach_trace (t, buf, level). This function attaches a trace to a buffer. The level value indicates the level of the trace stack at which the trace is attached to the buffer. The level field may contain the value ROOT if the entire record is attached, or may contain the value LEAF if only the active field is attached.

t = copy_trace (t1). This function obtains a new trace (by calling the get_trace function) and makes the new trace a copy of t1. If the original trace was attached to a buffer, then the new trace will also be attached to the buffer.

encode_str (str, buf). This function converts a C string into a GENESIS string. The value "str" is a pointer to the C string. The function places the GENESIS string into the buffer "buf".

free_trace(t). This function places a trace in the set of available traces.

get_trace. If there are available traces, this function will return one of the available traces. If there are no available traces, this function creates a new trace.

t = init_trace (Fdt_row). This function provides a new trace which is rooted at the element which is in the row "Fdt_row" of the Fdt.

print_trace (t, how_much). This function prints a copy of a trace onto the standard output. The value of "how_much" determines how much information is printed. If "how_much" is set to VERBOSE, then the entire trace is printed. If "how_much" is set to TERSE, then an abbreviated output is printed.

refresh_trace (t). This function aligns a trace by calculating the offsets to each field of the current record in the buffer that the trace is attached to. This function is used in the following situations:

1. After a trace has been attached to a buffer.
2. When a new record is read into the buffer that the trace is attached to.

3. When a record in a buffer is updated using a trace and there are other traces attached to the same buffer, then the other traces should be refreshed.

reset_status. This function resets the value in the status field of a trace to OKAY. This function is used after an error condition has been detected and appropriate action has been taken.

set_tr. This function returns a trace to a field corresponding to a complete field name such as file.field.subfield.etc.

NAVIGATIONAL FUNCTIONS

down (t, n). This function repositions a trace to the nth child of the active field of the trace.

down2 (t, n). This function is similar to down except that it is used for repeating groups.

field_left (t). This function repositions a trace to its left sibling.

left (t). This function is similar to field_left except that it is used for repeating groups.

field_right. This function repositions a trace to its right sibling.

right. This function is similar to field_right except that it is used for repeating groups.

restore. This function repositions a trace to the level stored in the "start level" of the trace. This level represents the level of the trace that is attached to the buffer.

skip(t). This function repositions a trace to its nth sibling.

up (t). This function repositions a trace to its parent field.

up2 (t). This function respositions a trace to its grandparent field.

INFORMATION FUNCTIONS

num = count (t). This function returns a count of the number of children of the active field of a trace.

index = ft (t). This function returns the index into the Fdt of the active field of a trace.

ln = len (t). This function returns a count of the number of bytes in the active field of a trace.

addr = loc (t). This functions returns the location in main memory of the active field of a trace.

boolean = rwok (t). This function returns the value TRUE if a trace is virtually positioned. Otherwise, this function returns FALSE. If a trace is virtually positioned, then it cannot read from or write to a field.

stat = status (t). This function returns the current value of the "status" field of the trace.

I/O FUNCTIONS

ad (t, buf). This function is used to add a new element to a repeating group. The trace points to the repeating group, and the buffer contains the new element to be added.

dl (t, n). This function is used to delete an element of a repeating group. The nth element of the repeating group pointed to by the trace is deleted.

filed_copy (t1, t2). This function copies the value of the active field of t1, into the active field of t2.

mk (t, buf). This function is used to set up a buffer to create a new record to be added to the database. The function attaches the trace to the buffer and puts an image of the trace into the buffer.

rd (t, buf). This function copies the active field of the trace into the buffer.

rep (t, buf). This function copies the value in the buffer to the active field of the trace.

The characteristics described in this chapter make the GENESIS Record Manager a powerful tool for manipulating the fields within a record. The key characteristic of the GENESIS Record Manager that makes it useful for this thesis is that it supports group attributes which are required for nested relations. The next chapter describes an algorithm for converting a nested relational algebra query to GENESIS Trace Manager commands.

V. Algorithm

This chapter describes the design of the algorithm which translates the nested relational algebra expressions produced by the SQL/NF translator into the GENESIS Trace Manager commands for executing the query. This chapter is divided into eight sections. Section 5.1 describes design decisions that were made during the development of the algorithm. Section 5.2 defines terms that will be used in this chapter. Section 5.3 defines a database that will be used in describing the algorithm. Section 5.4 describes the structure of nested relational algebra queries. Section 5.5 describes intermediate data structures that are used in processing the query. Section 5.6 presents the algorithm. Section 5.7 validates that the algorithm correctly translates nested relational algebra queries and provides examples of nested relational algebra queries, intermediate data structures, and outputs. Section 5.8 analyzes the performance of the algorithm in terms of the order, or Big O, of the parts of the algorithm.

5.1 Algorithm Design

This section describes some of the design decisions made during the development of the algorithm. One of the design decision was to divide the algorithm into two steps. The first step consists of building an intermediate data structure to represent the query as a parse tree, and the second step consists of processing the intermediate data structure to produce the output of the query. The reason for using two steps was to divide the complex process of translation into two simpler processes. A parse tree was used as an intermediate data structure because it provides a concise graphical representation of the query. The parse tree was designed as an n-ary tree because the nested relational algebra uses n-ary operators.

One of the problems faced in this thesis was how to evaluate the algorithm in terms of correctness and performance. Because it is not possible to test all possible nested relational algebra queries a method had to be devised to test a representative sample of queries. The validation approach used in this thesis consisted of two steps. The first step was analysis of nested relational algebra queries to determine the types of expressions that can occur within a query. The second step was to show that the algorithm correctly translates sample nested relational algebra queries containing each of the types of expressions.

The performance of the algorithm was evaluated by analysing the order, also referred to as Big O [15], of the algorithm. The approach used in the performance analysis was to determine the order for each of the parts of the algorithm. This approach has the advantage that is simplifies the performance analysis and it points out the parts of the algorithm that will limit performance for queries in general. In addition, this approach simplifies the analysis of a specific query because it allows each part of the query to be examined to determine its effect on the overall performance of the query.

The GENESIS Trace Manager has the limitation that it does not include all of the functions necessary to process a nested relational algebra query. The GENESIS Trace Manager is limited to manipulating fields within a record located in a buffer in primary memory. The GENESIS Trace Manager does not itself contain any facilities for the following:

1. Reading a record from a database and putting it into a buffer.
2. Performing boolean (AND, OR, NOT), predicate(<, >, =, <=, >=, <>), or aggregate (SUM, AVG, MIN, MAX, COUNT) functions.
3. Formatting the printed output of a query.

Because the GENESIS Trace Manager does not include these facilities, the algorithm does not generate GENESIS commands to perform these functions. The algorithm uses the phrase "read a record" to indicate that a record should be read from the database into a buffer in memory. When the GENSIS DBMS is completed, this part of the algorithm needs to be modified to include the commands for reading a record. The algorithm uses the GENESIS command "print_trace" to output data. This command enables the algorithm to output all the data for a query, but does not format the data into a report.

5.2 Definitions

The description of the algorithm in this chapter will include references to operations involving traces, buffers and records. In order to clarify these concepts, the following definitions are provided.

A trace is a data structure used to store the path from the root field in the record to the active field. A buffer is an area in memory used to store a record that has been read in from secondary storage. In the following discussion, 't' represents a trace, and 'b' represents a buffer.

When the phrase "define a trace" is used in the algorithm, it indicates the following GENESIS commands should be performed:

1. t = init_trace()
2. set_tr(fileid, field_name, t)
3. attach_trace(t, b, level)

The first step gets a new trace, the second step sets the trace to the specified field, and the third step attaches the trace to a buffer.

When the phrase "read a record" is used in the algorithm, it indicates that a record should be read into the buffer associated with a specified trace and the GENESIS command "refresh_trace" should be performed. The refresh_trace command realigns the trace with respect to the new record that was read into the buffer.

5.3 Database Description

In the description of the algorithm, numerous examples will be used. These examples will use the database schema shown in Figure 25. GENESIS represents this database schema as an

```
DATABASE dept_db {
TYPES
        PARTSET = {
        part          ARRAY [2] OF CHAR;
        };
PERSON = {
        name          ARRAY [10] OF CHAR;
        dob           ARRAY [8] OF CHAR;
        };
EMPLOYEE = {
        empno         ARRAY [2] OF CHAR;
        name          ARRAY [10] OF CHAR;
        sal           ARRAY [7] OF CHAR;
        mgr           ARRAY [2] OF CHAR;
        children      RPG OF PERSON;
        };
FILES
/* department file */
dept {
        dno           ARRAY [2] OF CHAR;
        dname         ARRAY [10] OF CHAR;
        loc           ARRAY [10] OF CHAR;
        emp           RPG OF EMPLOYEE;
        usage         RPG OF PARTSET;
        };
/* supply file */
supply {
        supplier      ARRAY [2] OF CHAR;
        supplies      RPG OF PARTSET;
        };
}. /* end of schema */
```

Figure 25. Database Schema

inverted tree structure as shown in Figure 26. The sample data that will be used in the examples is shown in Figure 27. In order to save space in the Supply relation in Figure 27, the parts are listed separated by commas instead of in the vertical format used in the Department relation.

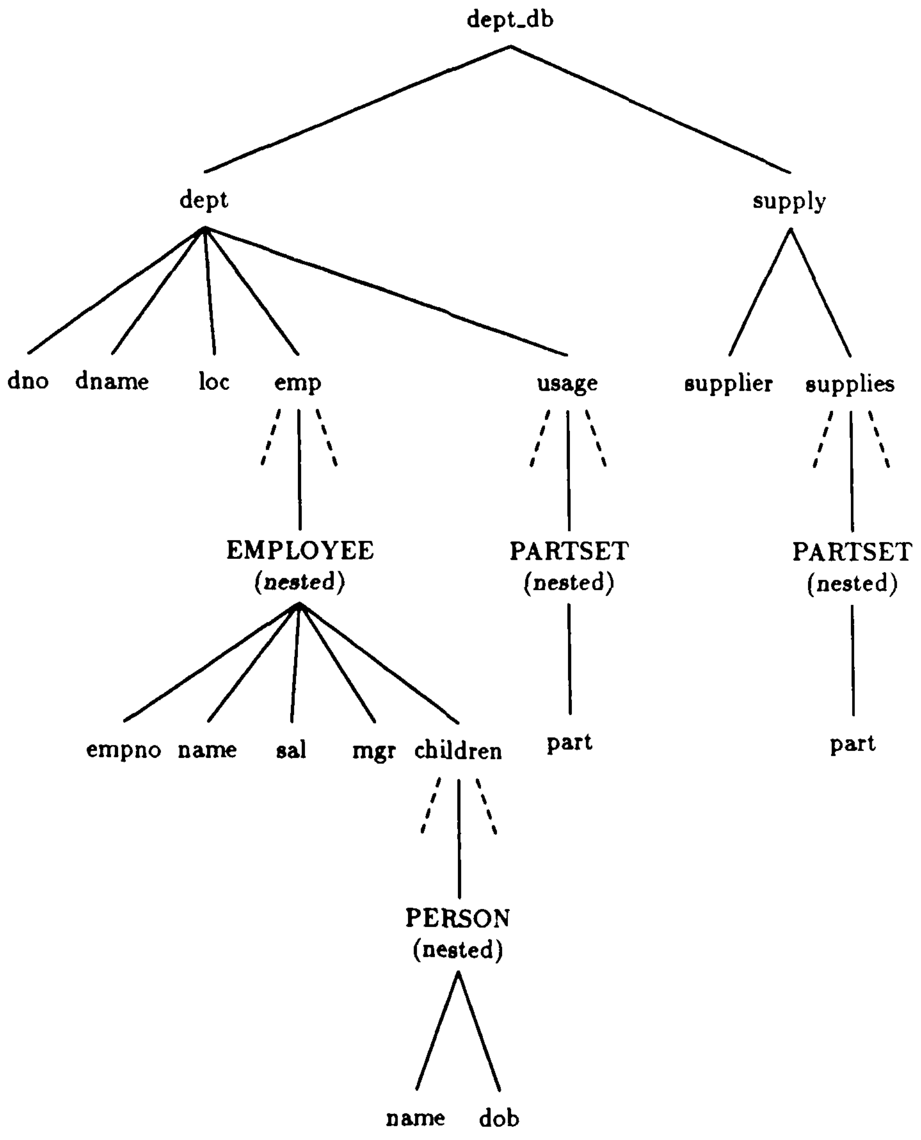


Figure 26. Tree Structure Representation of Database

Department

DNO	DNAME	LOC	EMP						USAGE
			EMPNO	NAME	SAL	MGR	CHILDREN		PART
							NAME	DOB	
10	Accounting	New York	11	John Smith	$30,000	19	Jim Smith	11/12/81	11
							Mary Smith	09/03/83	12
			12	Pat Green	$33,000	19	Mary Green	28/02/79	13
							Pete Green	21/07/81	14
			13	Jim Jones	$35,000	19	John Jones	02/03/84	15
							Pat Jones	04/05/85	
20	Finance	New York	21	Bob Harris	$43,000	29	Jim Harris	06/07/71	21
							Pam Harris	08/09/72	22
			22	Mary Hill	$47,000	29	Pat Hill	13/12/73	23
							Jerry Hill	23/11/75	24
			23	Tim Taylor	$53,000	29	Tom Taylor	19/07/76	25
							Eve Taylor	18/06/78	
30	Shipping	Dallas	31	Joe Swartz	$28,000	39	Jim Swartz	02/01/85	31
							Pam Swartz	04/03/86	32
			32	Patty Swan	$26,000	39	Frank Swan	31/01/84	33
							Sally Swan	28/02/86	34
			33	Terry Bell	$52,000	39	Cindy Bell	24/05/81	35
							Sam Bell	22/06/82	
40	Research	Dallas	41	Dave Hamil	$39,000	49	Bob Hamil	13/10/67	41
							Pat Hamil	18/12/68	42
			42	Ed Lawson	$41,000	49	Joe Lawson	24/02/68	43
							Jan Lawson	12/07/70	44
			43	Tim Miller	$36,000	49	Bob Miller	27/01/72	45
							Pat Miller	14/07/73	
50	Personnel	New York	51	Mike Owens	$44,000	59	Bill Owens	22/09/81	51
							Mary Owens	19/11/83	52
			52	Bob Jones	$38,000	59	Tim Jones	14/01/76	53
							Pam Jones	17/07/77	54
			53	Fred Tate	$44,000	59	John Tate	05/04/78	55
							Molly Tate	09/08/79	

Supply

SUPPLIER	SUPPLIES
	PART
National	11, 12, 13, 14, 15, 21, 22, 23, 24, 25, 31, 32, 33, 32, 35
Titan	31, 32, 33, 34, 35, 41, 42, 43, 44, 45, 51, 52, 53, 54, 55
Wilson	21, 22, 23, 24, 25, 41, 42, 43, 44, 45

Figure 27. Sample Data

5.4 Structure of a Nested Relational Algebra Query

A nested relational algebra query produced by the SQL/NF translator may consist of the following basic operators: projection (PJ), selection (SL), Cartesian product (CP), union (UN), intersection (IN), difference (DF) and functional evaluation (FN).

The PJ statement may contain the following structures:

1. A field name, i.e., a dot expression describing the full path name of a field. For example, "dept.emp.name" would refer to the "employee's name" attribute in the "dept" file.
2. A new user supplied name appearing in the query. This can be recognized by the fact that it ends with a colon. For example, in the statement:

   PJ[department: dept.dname]

   the word "department" is a new user supplied name for "dept.dname".
3. A PJ-SL-CP block. For example, the following query contains an inner PJ-SL-CP block:

```
PJ[dept.name, children: (PJ[dept.emp.children.name]
                         SL[dept.name ="Accounting"]
                         CP[dept.emp.children])]
```

4. One of the following functions, UNION, INTERSECTION, MINUS, AVG, MIN, MAX, SUM, COUNT, NEST, UNNEST.

The SL statement may contain the following structures:

1. A predicate which may contain the following operators: ANY, ALL, [IS] IN, NOT IN, EXISTS, NOT_EXISTS, CONTAINS, DOES NOT CONTAIN, =, <>, <, >, <=, or >=.
2. A boolean expression consisting of two or more predicates connected with AND, OR, or NOT.
3. A PJ-SL-CP block.

The CP statement may contain:

1. File names.

2. A query expression.

The UN function, IN function, and DF function operate on two PJ-SL-CP blocks. UN produces a relation which contains tuples in either of PJ-SL-CP blocks. IN produces a relation which contains only tuples that occur in both PJ-SL-CP blocks. DF produces a relation which contains tuples in the first PJ-SL-CP block that are not in the second PJ-SL-CP block.

The FN statement is used in representing nest expressions and aggregate functions which occur in the PJ statement. There are three forms of FN statements which are defined as follows [24, pages 90–91]:

$$\mathrm{FN}[F(B);\phi]R = \mathrm{CP}[R, R^{F(B)}]$$

with F(B) being a single attribute and $R^{F(B)}$ being a relation with only one tuple, specified by applying F(B) to R.

$$\mathrm{FN}[F(B);A]R = \cup_{t\epsilon R}\ \mathrm{FN}[F(B);\phi]\ (\mathrm{SL}[A = T.A]R)$$

where A is a set of attributes, t is a tuple of R, and t.A gives the values taken by A in R.

$$\mathrm{FN}[F1(B1), F2(B2), ..., Fn(Bn); A]R = \mathrm{FN}[F_1(B_1);A](FN[F_1(B_1);A](...F_n(B_n);A]R...))).$$

The FN operation extends the relation R to include new attributes that correspond to F(B). An example of an FN statement from [3, page 328] is shown in Figure 28.

5.5 *Intermediate Data Structures.*

The conversion of a nested relational algebra query into GENESIS commands occurs in two phases. The first phase consists of representing the query as an inverted tree structure and the second phase consists of processing the tree structure to produce the query result. This section describes the tree structure used to represent the query.

There are seven basic types of expressions in nested relational algebra that need to be represented in the tree structure. These seven expressions are:

ENO	SAL	DNO
1	100	1
2	150	2
3	130	1
4	170	2
5	120	2
6	160	1
7	170	3

Relation EMP (ENO, SAL, DNO)

ENO	SAL	DNO	AVG(SAL)
1	100	1	130
2	150	2	135
3	130	1	130
4	170	2	170
5	120	2	135
6	160	1	130
7	170	3	170

Relation Produced as result of FN[AVG(SAL), DNO] EMP

Figure 28. Example of FN Statement

1. PJ-SL-CP block
2. Set operator expressions involving (UN, IN, DF)
3. Predicate expressions involving (=, <, >, <=, >=, <>, CONTAINS, BETWEEN AND, IN, EXISTS)
4. Boolean expressions involving (AND, OR, NOT)
5. Aggregate expressions involving (MAX, MIN, AVG, SUM, COUNT)
6. NEST and UNNEST expressions
7. FN expressions

The basic tree structure for a PJ-SL-CP block consists of one branch for each of the PJ, SL, and CP structures. For example, the query

```
PJ[expression A]
SL[expression B]
CP[expression C]
```

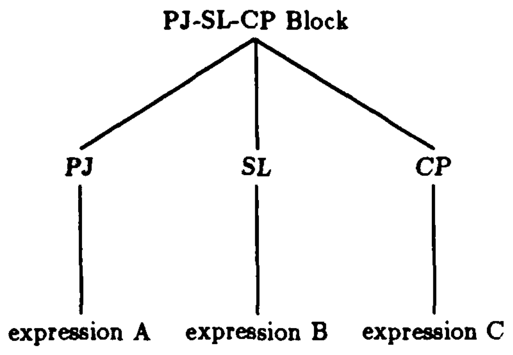


Figure 29. PJ-SL-CP Structure

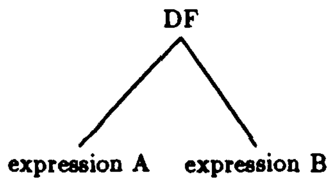


Figure 30. DF Structure

would have the structure shown in Figure 29.

If the query contains a set operator (UN, IN, or DF), then the tree contains a node for the set operator and a branch for each expression. For example, the expression

DF[(expression A), (expression B)]

would have the *structure shown in Figure 30*.

A predicate expression is represented by a node for the predicate (=, <, >, <=, >=, <>, CONTAINS, BETWEEN AND, IN, EXISTS) and a branch for each operand. The EXISTS predicate has only one operand, the BETWEEN AND predicate has three operands, and the other predicates

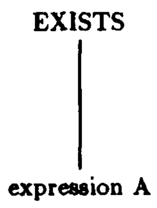


Figure 31. EXISTS Structure

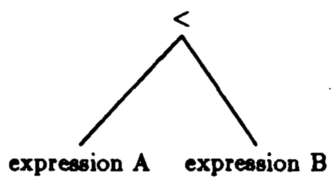


Figure 32. Less Than Structure

each have two operands. The following examples show the tree structure for predicates with one, two, and three operands. The predicate

EXISTS(expression A)

would have the structure shown in Figure 31.

The predicate

(expression A) < (expression B)

would have the structure shown in Figure 32.

The predicate

(expression A) BETWEEN (expression B) AND (expression C)

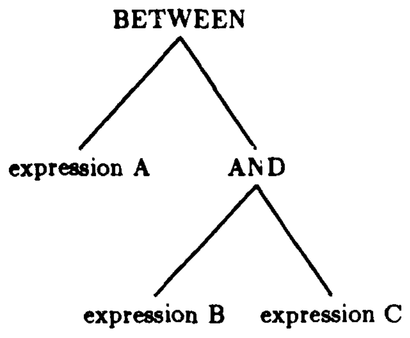


Figure 33. BETWEEN Structure

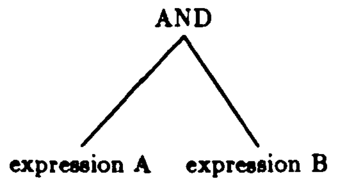


Figure 34. AND Structure

would have the structure shown in Figure 33.

A boolean expression is represented by a node for the boolean operator (AND, OR, NOT) and a branch for each operand. For example, the expression

(expression A) AND (expression B)

would be represented by the structure shown in Figure 34.

An aggregate expression is represented by a node for the aggregate operator (MAX, MIN, AVG, SUM, COUNT) and a branch for the operator. For example, the expression

AVG(expression A)

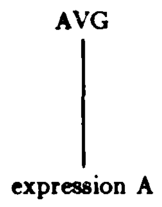


Figure 35. AVG Structure

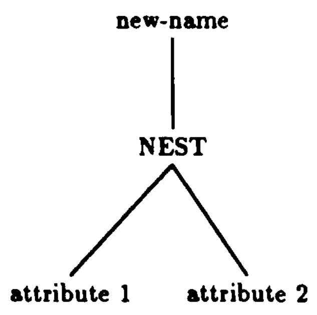


Figure 36. NEST Structure

is represented by the structure shown in Figure 35.

A NEST expression is represented by a node for the NEST operator, a node for the name of the new nested structure, and a branch for each attribute. For example the expression

new-name: NEST(attribute 1, attribute 2)

would be represented by the structure shown in Figure 36.

An UNNEST expression is represented by a node for the UNNEST operator, a node for the name of the UNNEST structure, and a branch for each attribute. For example, the expression

new-name: UNNEST(attribute 1, attribute 2, attribute 3)

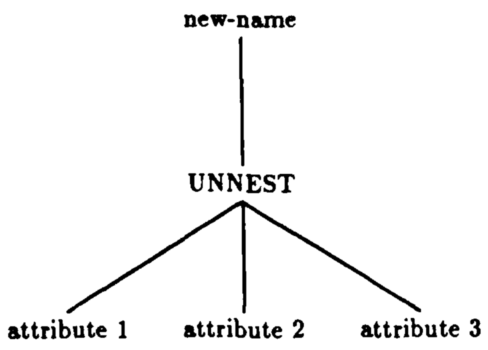


Figure 37. UNNEST Structure

would be represented by the structure shown in Figure 37.

An FN expression is represented by a node for the FN operator, a node for the new name associated with the FN expression and a node for each operator in the expression. For example, the expression

FN[new name: expression A; expression B]

would be represented by the structure shown in Figure 38.

An actual query would involve a combination of the structures described above. For example the query

```
PJ[dept.name, dept.loc]
SL [(dept.loc = "Chicago") AND (dept.usage CONTAINS
    dept.usage.part = 12)]
CP[dept]
```

would be represented by the structure shown in Figure 39.

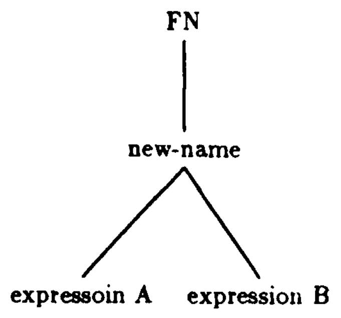


Figure 38. FN Structure

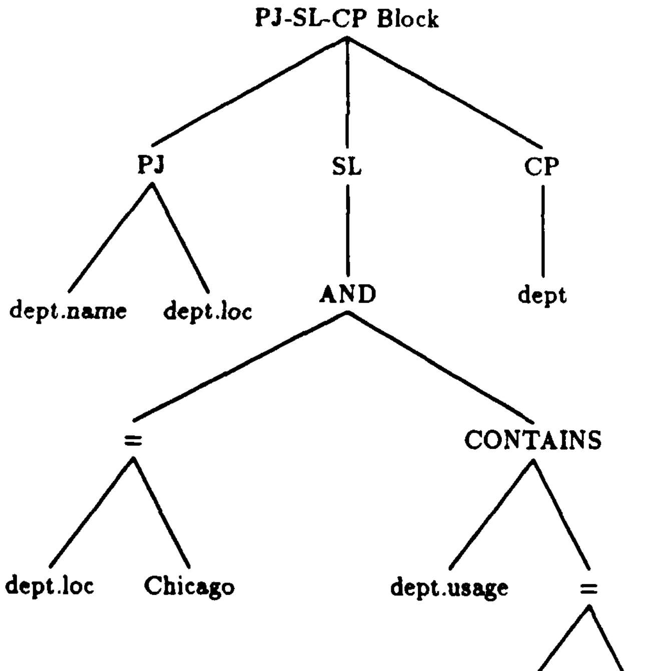


Figure 39. Query Structure

5.6 Algorithm

This section describes the algorithm for converting nested relational algebra queries to GENESIS Trace Manager commands. The algorithm to convert the nested relational algebra to GENESIS commands consists of two phases. The first phase is building an intermediate data structure to represent the query and the second phase processes the data structure to produce the output of the query. The first phase consists of the following steps:

1. For the main PJ-SL-CP block in the query, define a tree structure containing a branch for each of the PJ, SL, and CP clauses.

2. For each structure in the project statement

   2a. If the structure is a simple field name, then add a branch to the PJ node and define a trace PJ_ti where 'i' is an index starting at one. For example, if the project statement is:

   PJ dept.name dept.loc

   then the following traces would be defined:

   PJ_t1 = a trace pointing to [dept.name]

   PJ_t2 = a trace pointing to [dept.loc]

   2b. If the structure is a new user supplied name for a relation, then add a branch to the PJ node for the new name and add a branch to the new name node for each field in the new name. Also, define a trace PJ_ti for each field in the relation as in 2a above.

   2c. If the structure is a NEST expression, then it will be preceded by a new user name. To process the NEST expression, add a node for the new name, add a node to the new name node for the NEST expression, and add a branch to the NEST node for each field in the NEST expression. Also, define a trace PJ_ti for each field in the NEST expression. If the structure is an UNNEST expression, then add a node for the UNNEST expression and add a branch to the UNNEST node for each field in the UNNEST expression. Also, define a trace PJ_ti for each field in the UNNEST expression.

2d. If the structure is a set operator expression(UN, IN, DF), then add a node for the set operator and add a branch to the node for each operand of the set operator. Then process each operand as in step 1 above.

2e. If the structure is an aggregate expression(AVG, MIN, MAX, SUM, or COUNT), then define a node for the operator and add a branch to the node for the operand. Then process each operand as in step 1 above.

2f. If the structure is a PJ-SL-CP block then process it starting in step 1 above.

3. For each structure in the Cartesian product statement:

3a. If the structure is a file name, then add a branch to the CP node and define a trace CP_ti, where 'i' is an index starting at one. For example, if the Cartesian product statement is:

CP dept supply

then the following traces would be defined:

CP_t1 = a trace pointing to [dept]

CP_t2 = a trace pointing to [supply]

3b. If the structure is a PJ-SL-CP block, then it is preceded by a user defined name. Add a branch to the CP node for the new name, add a node to the new name node for the PJ-SL-CP block; add PJ, SL, and CP branches to the PJ-SL-CP block node; and process the PJ-SL-CP block as in step 1 above.

4. For each structure in the select clause:

4a. If the structure is a predicate expression then add a node for the predicate, add a branch to the predicate node for each predicate operand, and process each operand as in step 1 above.

4b. If the structure is a boolean expression then add a node for the boolean operator, add a branch to the boolean node for each boolean operand, and process each operand as in step 1 above.

4c. If the structure is a PJ-SL-CP block then add a node for the block and process the block as in step 1 above.

5. If the structure is a set operator expression, then add a node for the set operator, add a branch to the operator node for each operand, and process each operand as in step 1 above.

6. If the structure is an FN expression, then add a node for the FN operator, add a branch to the FN node for the new name, add a branch to the new name node for each operand, and process each operand as in step 1 above.

This completes the first phase of the algorithm. At this stage in the algorithm a tree structure has been defined to represent the query and a trace has been assigned to each leaf in the tree that represents a field name. Each leaf of the PJ branch of the tree is a field name (i.e. dept.dname). Each leaf of the SL branch is either a field name or a constant (i.e. 12, "Chicago"). Each leaf of the CP branch is a relation name.

The second phase of the algorithm is processing the tree structure to produce the query result. This consists of the following steps:

The root node of the tree will be either a set operator or a PJ-SL-CP block. If the root node of the tree is set operator, then perform algorithm Set_op_process. If the root node is a PJ-SL-CP block, then perform algorithm PJ-SL-CP_process.

The Set_op_process consists of the following steps:

1. Process each branch of the set operator node using PJ-SL-CP_process.
2. If the set operator is UN, then print tuples that are in the results of either of the PJ-SL-CP blocks. If the set operator is DF, then print tuples that are in the first PJ-SL-CP block but not in the second PJ-SL-CP block. If the set operator is IN, then print tuples that are in the results of both of the PJ-SL-CP blocks.

The PJ-SL-CP_process consists of the following steps:

1. For each branch of the CP node, create a buffer CP_bi, where 'i' is an index starting at one.
2. Read the first record of each CP branch into the corresponding buffer.
3. Process the SL branch by performing SL_process.
4. If the SL node has been assigned a value of false by SL_process, then the records currently in the CP buffers do not meet the criteria in the SL branch and none of the traces in the PJ branch are printed. If the SL node has been assigned a value of true, then perform PJ_process.
5. If there are more records in the database, then read the next record into the CP buffer and return to step 3.

The goal of the SL_process is to determine if the records currently in the CP buffers meet the criteria in the SL branch of the query. The SL_process will result in a value of true if all the criteria in the SL_branch are met, and will result in a value of false if any of the criteria are not met. This process starts at the lower levels of the SL branch and works up to the top of the SL branch because the truth value of the lower nodes must be determined before the truth value of the upper nodes can be determined. The SL_process consists of the following steps:

1. Each leaf node of the SL branch is either a field name or a constant. For each leaf node that is a field name:

   1a. Attach the node's trace to the CP buffer corresponding to the first part of the field name.

   1b. Refresh the trace.

2. Starting at the lowest level of the SL branch, process each predicate and boolean expression by comparing the operands in the expression using the operator in the predicate or boolean node. If the expression evaluates to true, then assign a value of true to the predicate or boolean node. If the expression evaluates to false, then assign a value of false to the node.

3. Repeat the process in step 2 for each successive level of the SL branch until the SL node has been assigned a value of true or false.

The goal of the PJ_process is to produce as output a relation containing all the fields in the PJ branches. The PJ branches may consist of field names or new user supplied names. The PJ_process consists of performing the following steps for each branch of the PJ node:

1. If the branch consists of a field name, then perform the following steps:

   1a. Attach the field's trace to the corresponding CP buffer

   1b. Refresh the trace

   1c. Perform Print_trace

2. If the branch consists of a new name, then perform New_name_process.

The New_name_process consists of the following steps:

1. If the new name consists of a field name:

   1a. Attach the field's trace to the corresponding CP buffer

   1b. Refresh the trace

   1c. Perform Print_trace

2. If the new name consists of an aggregate function, then perform Aggregate_process.

3. If the new name consists of a PJ-SL-CP block then perform PJ-SL-CP_process.

4. If the new name consists of a set operator, then perform Set_op_process.

5. If the new name consists of a NEST operator, then perform NEST_process.
6. If the new name consists of an UNNEST operator, then perform UNNEST_process.

The goal of the Aggregate_process is to evaluate and output the result of an aggregate expression. An aggregate expression consists of an aggregate function (AVE, MIN, MAX, SUM, COUNT) and an operand. The operand may be a field name, or a user supplied name. The Aggregate_process consists of the following steps:

1. If the operand is a field name, then perform the following steps:
   - 1a. Attach the trace to the corresponding CP buffer
   - 1b. Read the first record into the CP buffer
   - 1c. Refresh the trace
   - 1d. Perform the aggregate function on the field pointed to by the trace.
   - 1e. If there are more records, then read the next record into the CP buffer and return to step 1c.
2. If the operand is a user supplied name, then perform New_name_process and go back to step 1.

The goal of NEST_process is to create a new relation corresponding to the NEST expression. The NEST_process consists of the following steps:

1. Create a list to keep track of which records have been processed.
2. Read the first record into the CP buffers .
3. Save a pointer to the current record and add this record number to the list of records that have been processed.
4. Attach the traces to the CP buffers.
5. Refresh the traces.
6. Save the values of the fields pointed to by each trace in the expression.

7. If the values that are currently pointed to by all non-NEST traces match the values saved in step 6, then for each trace in the NEST expression perform Print_trace and add this record number to the list of records that have been processed.
8. If there are more records, then read the next record into the CP buffers, refresh the traces and return to step 7.
9. Restore the pointers saved in step 3.
10. If there are more records, then find the next record that is not in the list of processed records, read the record into the CP buffers and return to step 3.

The goal of UNNEST_process is to create a new relation corresponding to the UNNEST expression. The UNNEST_process consists of the following steps:

1. Read the first record into the CP buffers.
2. Attach the traces to the CP buffers.
3. Refresh the traces.
4. For each trace in the expression perform Print_trace.
5. If there are more nested values in the NEST expression, then update the traces in the NEST attributes and for each trace in the expression, perform Print_trace.
6. If there are more records, then read the next record into the CP buffers and return to step 2.

The Print_trace_process uses the count(t) function which returns the number of children of the field pointed to by trace t. If the count is equal to zero, then there are no children and only the field itself is printed. If the count is greater than zero, then the field has children (i.e., it is a nested structure), and each child needs to be printed. In this case, a trace is established for each child field. If there is more than one level of nesting, then the above procedure is applied recursively until all children have been printed. The Print_trace_process consists of the following steps:

1. Determine if the trace is an atomic attribute or a nested attribute by using the GENESIS statement:

   children = count(t)

2. If the trace has no children, then it is an atomic attribute and is printed.

3. If the trace has children it is a nested attribute, and a trace needs to be defined for each child. To define new traces for the nested attribute, perform the following steps for each child:

   3a. t_new = copy_trace(t)

   3b. down (t_new, n)

   Statement 3a returns a new trace variable "t_new" which is attached to the same buffer as t and points to the same field. In statement 3b, n is the child number, with n=0 for the first child. Statement 3b repositions "t_new" to point to its nth child.

4. Because it is possible for there to be multiple levels of nesting, each of the new traces created in step 3 must be processed starting in step 1 above.

5.7 *Validation*

This section validates that the algorithm correctly evaluates nested relational algebra queries. The approach used for validation is to divide query expressions into groups, and show that the algorithm works correctly for a sample case from each group. This validation procedure uses the database schema described in Section 5.3 and the examples provided at the end of this section.

There are seven basic operators that may occur within a nested relational algebra query. These operators are projection (PJ), selection (SL), Cartesian product (CP), union (UN), intersection (IN), difference (DF), and functional evaluation (FN). In addition to these basic operators, nested relational algebra queries may include expressions involving predicate operators (=, <, >, <=, >=, <>, CONTAINS, BETWEEN AND, IN, EXISTS), boolean operators (AND, OR, NOT), aggregate operators (MIN, MAX, AVG, SUM, COUNT), the NEST operator, and the UNNEST operator. These operators can be used in a nested relational algebra query to form seven types of expressions:

1. PJ-SL-CP block
2. Set operator expressions involving (UN, IN, DF)
3. Predicate expressions involving (=, <, >, <=, >=, <>, CONTAINS, BETWEEN AND, IN, EXISTS)
4. Boolean expressions involving (AND, OR, NOT)
5. Aggregate expressions involving (MAX, MIN, AVG, SUM, COUNT)
6. NEST and UNNEST expressions
7. FN expressions

In order to validate the algorithm, example nested relational algebra queries will be described that include samples of each type of expression, and it will be shown that the algorithm produces the correct intermediate data structure and query result for each example. Table 40 provides a cross-reference for each example and the types of expressions involved in the example. The table shows that each of the types of expressions is represented in at least one of the examples. Each example nested relational algebra query shows the intermediate data structure and query result produced by the algorithm for the sample database schema in Section 5.3. Examination of the examples shows that the algorithm generates the correct intermediate data structure and query result for each example. The examples validate that the algorithm works correctly for queries that include each of the types of expressions.

Example 1:

```
PJ [ dept.dno, dept.dname, dept.loc ]
CP [ dept ]
```

EXAMPLE	1	2	3	4	5	6	7	8
PJ-SL-CP block	X	X	X	X	X	X	X	X
Set operator							X	
Predicate operator		X			X		X	
Boolean operator		X						
Aggregate operator			X				X	X
NEST operator				X				
UNNEST operator					X			
FN operator			X	X	X		X	X

Figure 40. Cross-reference of Operators used in Examples.

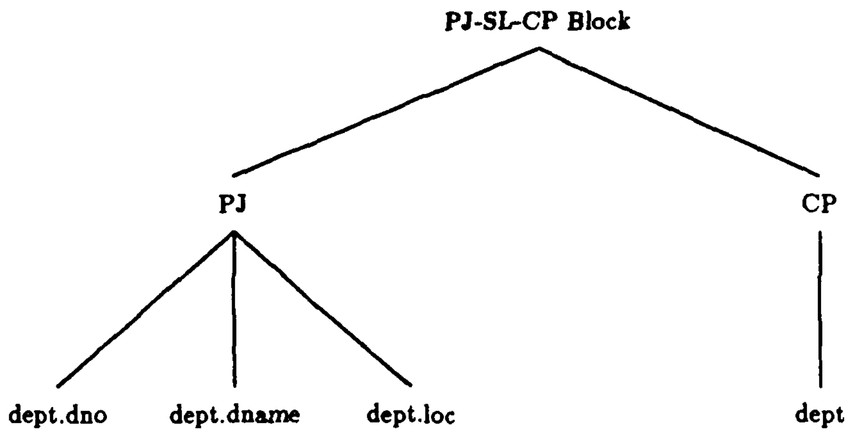


Figure 41. Example 1 Structure

The intermediate data structure for this query is shown in Figure 41. The output of the query is shown in Figure 42.

Example 2:

PJ [dept.dname, dept.emp.name, dept.emp.sal]
SL [dept.emp.sal > $40,000 AND dept.dname = "Finance"]
CP [dept]

The intermediate data structure for this query is shown in Figure 43. The output of the query is shown in Figure 44.

DNO	DNAME	LOC
10	Accounting	New York
20	Finance	New York
30	Shipping	Dallas
40	Research	Dallas
50	Personnel	New York

Figure 42. Example 1 Output

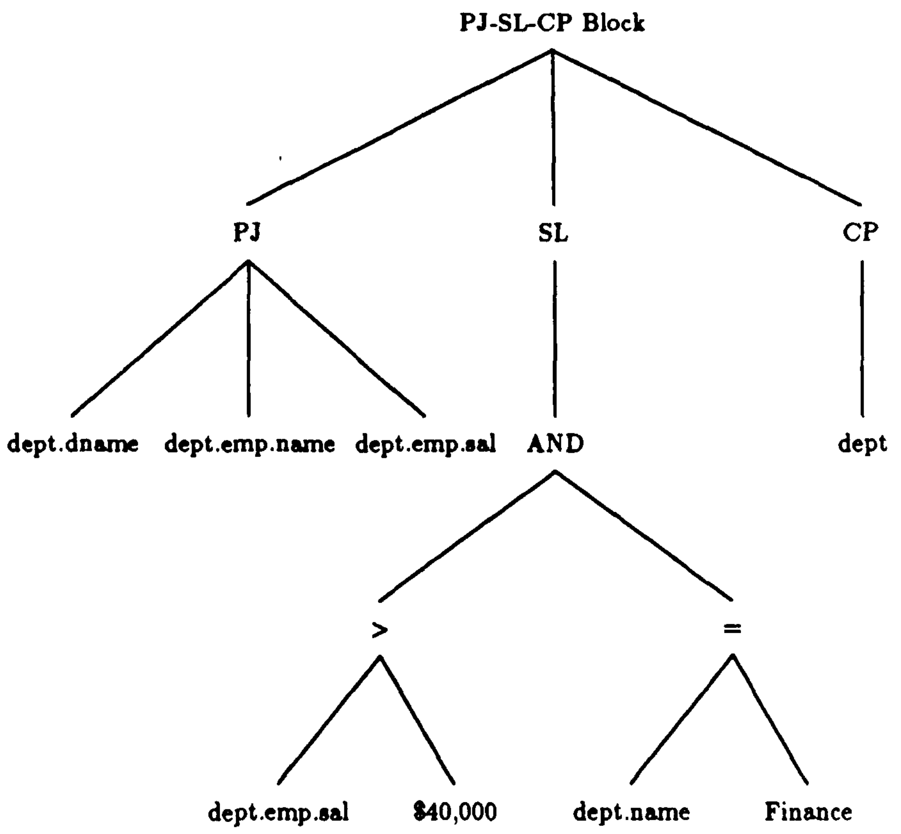


Figure 43. Example 2 Structure

DNAME	NAME	SAL
Finance	Bob Harris	\$43,000
	Mary Hill	\$47,000
	Tim Taylor	\$53,000

Figure 44. Example 2 Output

Example 3:

```
PJ [ dept.dname, AVERAGE-SAL: AVG(dept.emp.sal) ]
FN [ AVERAGE-SAL:AVG(dept.emp.sal; ϕ) ]
CP [ dept ]
```

The intermediate data structure for this query is shown in Figure 45. The output of the query is shown in Figure 46.

Example 4:

```
PJ [ dept.loc, LOC-INFO: NEST(dept.dno, dept.dname) ]
FN [ LOC-INFO: NEST(dept.dno, dept.dname) ]
CP [ dept ]
```

The intermediate data structure for this query is shown in Figure 47. The output of the query is shown in Figure 48.

Example 5:

```
PJ [ dept.emp.name, CHILD: UNNEST(dept.emp.children.name
                                   dept.emp.children.dob) ]
FN [ CHILD: UNNEST(dept.emp.children.name
                   dept.emp.children.dob) ]
SL [ dept.loc = "Dallas" ]
CP [ dept ]
```

The intermediate data structure for this query is shown in Figure 49. The output of the query is shown in Figure 50.

Example 6:

```
PJ [ dept.dname, PERSONNEL: ( PJ [ dept.emp.empno, dept.emp.name ]
                              CP [ dept.emp ]
CP [ dept ]
```

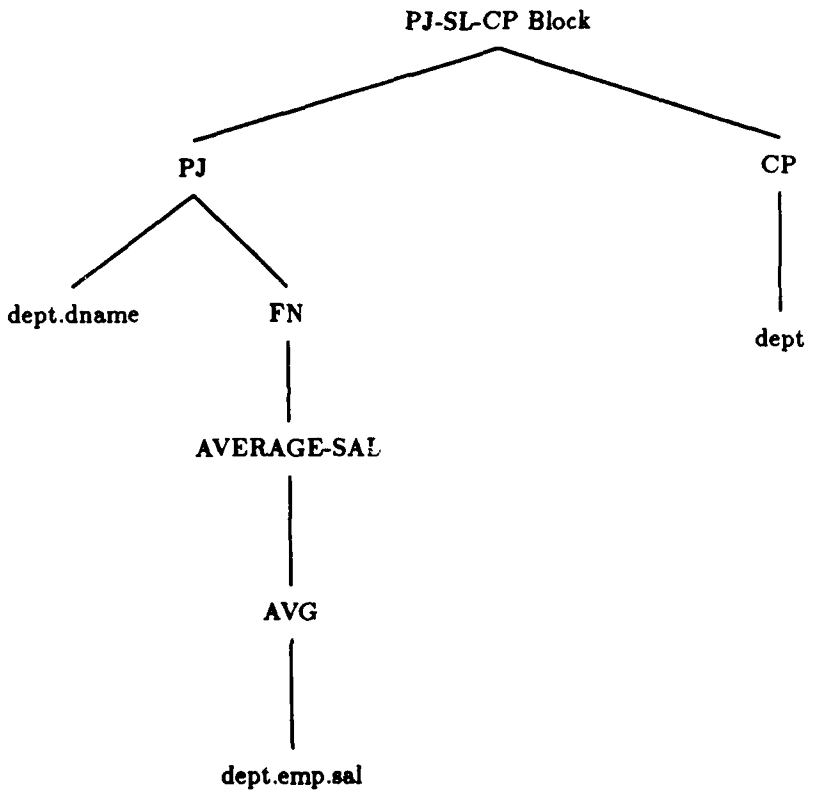


Figure 45. Example 3 Structure

DNAME	AVERAGE-SAL
Accounting	$32,667
Finance	$47,667
Shipping	$35,333
Research	$38,667
Personnel	$42,000

Figure 46. Example 3 Output

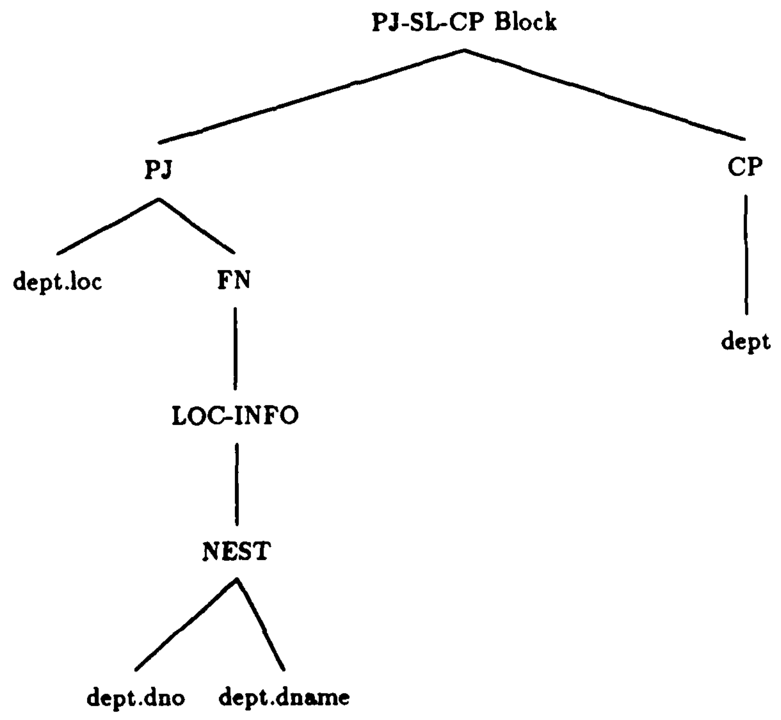


Figure 47. Example 4 Structure

LOC	LOC-INFO	
	DNO	DNAME
New York	10	Accounting
	20	Finance
	50	Personnel
Dallas	30	Shipping
	40	Research

Figure 48. Example 4 Output

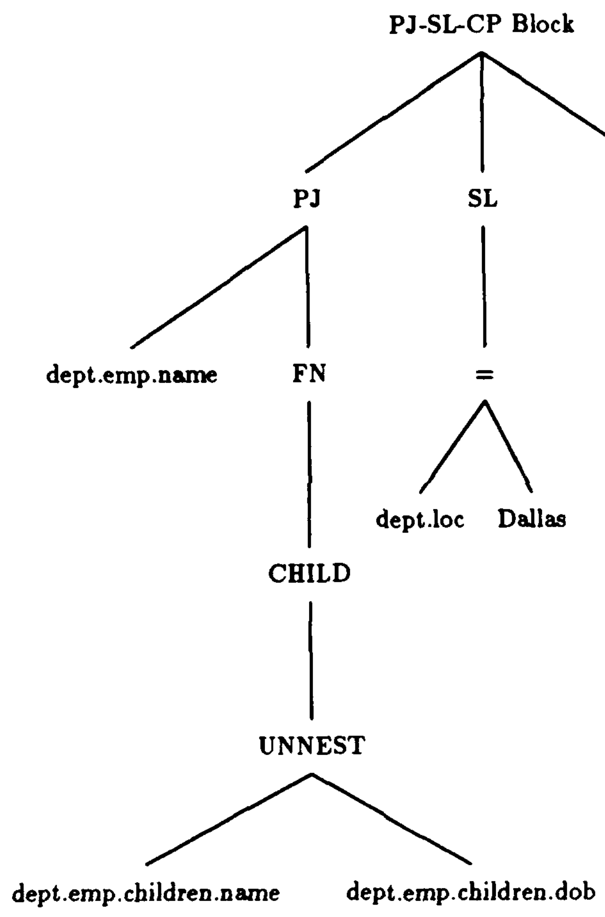


Figure 49. Example 5 Structure

EMP-NAME	CHILD-NAME	DOB
Joe Swartz	Jim Swartz	02/01/85
Joe Swartz	Pam Swartz	04/03/86
Patty Swan	Frank Swan	31/01/84
Patty Swan	Sally Swan	28/02/86
Terry Bell	Cindy Bell	24/05/81
Terry Bell	Sam Bell	22/06/82
Dave Hamil	Bob Hamil	13/10/67
Dave Hamil	Pat Hamil	18/12/68
Ed Lawson	Joe Lawson	24/02/68
Ed Lawson	Jan Lawson	12/07/70
Tim Miller	Bob Miller	27/01/72
Tim Miller	Pat Miller	14/07/73

Figure 50. Example 5 Output

The intermediate data structure for this query is shown in Figure 51. The output of the query is shown in Figure 52.

Example 7:

```
UN [ ( PJ [ dept.dno, dept.dname, dept.loc ]
       SL [ MIN-SAL: MIN(dept.emp.sal) < $30,000]
       FN [ MIN-SAL: MIN(dept.emp.sal) < $30,000]
       CP [ dept ] )
     ( PJ [ dept.dno, dept.dname, dept.loc ]
       SL [ MAX-SAL: MAX(dept.emp.sal) > $50,000]
       FN [ MAX-SAL: MAX(dept.emp.sal) > $50,000]
       CP [ dept ] ) ]
```

The intermediate data structure for this query is shown in Figure 53. The output of the query is shown in Figure 54.

Example 8:

```
PJ [ supply.supplier, SUPPLY-COUNT: COUNT(supply.supplies.part) ]
FN [ SUPPLY-COUNT: COUNT(supply.supplies.part) ]
CP [ supply ]
```

The intermediate data structure for this query is shown in Figure 55. The output of the query is shown in Figure 56.

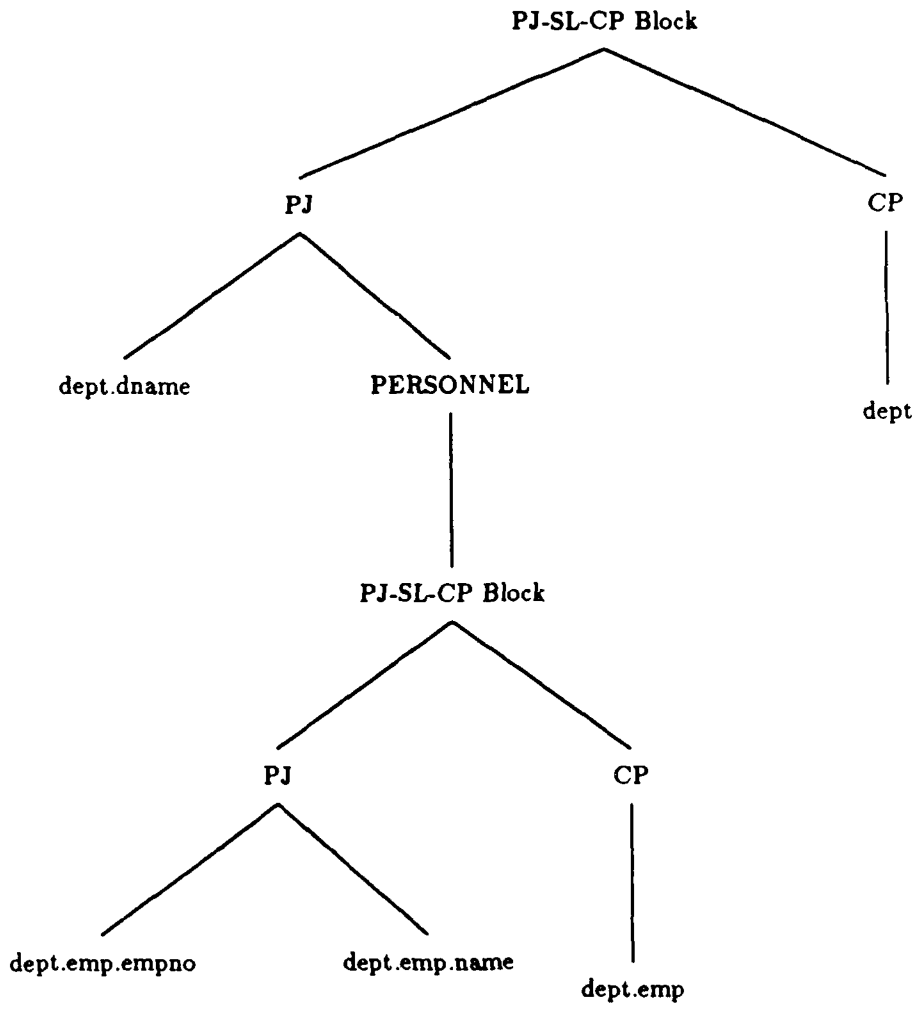


Figure 51. Example 6 Structure

DNAME	EMPNO	NAME
Accounting	11	John Smith
	12	Pat Green
	13	Jim Jones
Finance	21	Bob Harris
	22	Mary Hill
	23	Tim Taylor
Shipping	31	Joe Swartz
	32	Patty Swan
	33	Terry Bell
Research	41	Dave Hammil
	42	Ed Lawson
	43	Tim Miller
Personnel	51	Mike Owens
	52	Bob Jones
	53	Fred Tate

Figure 52. Example 6 Output

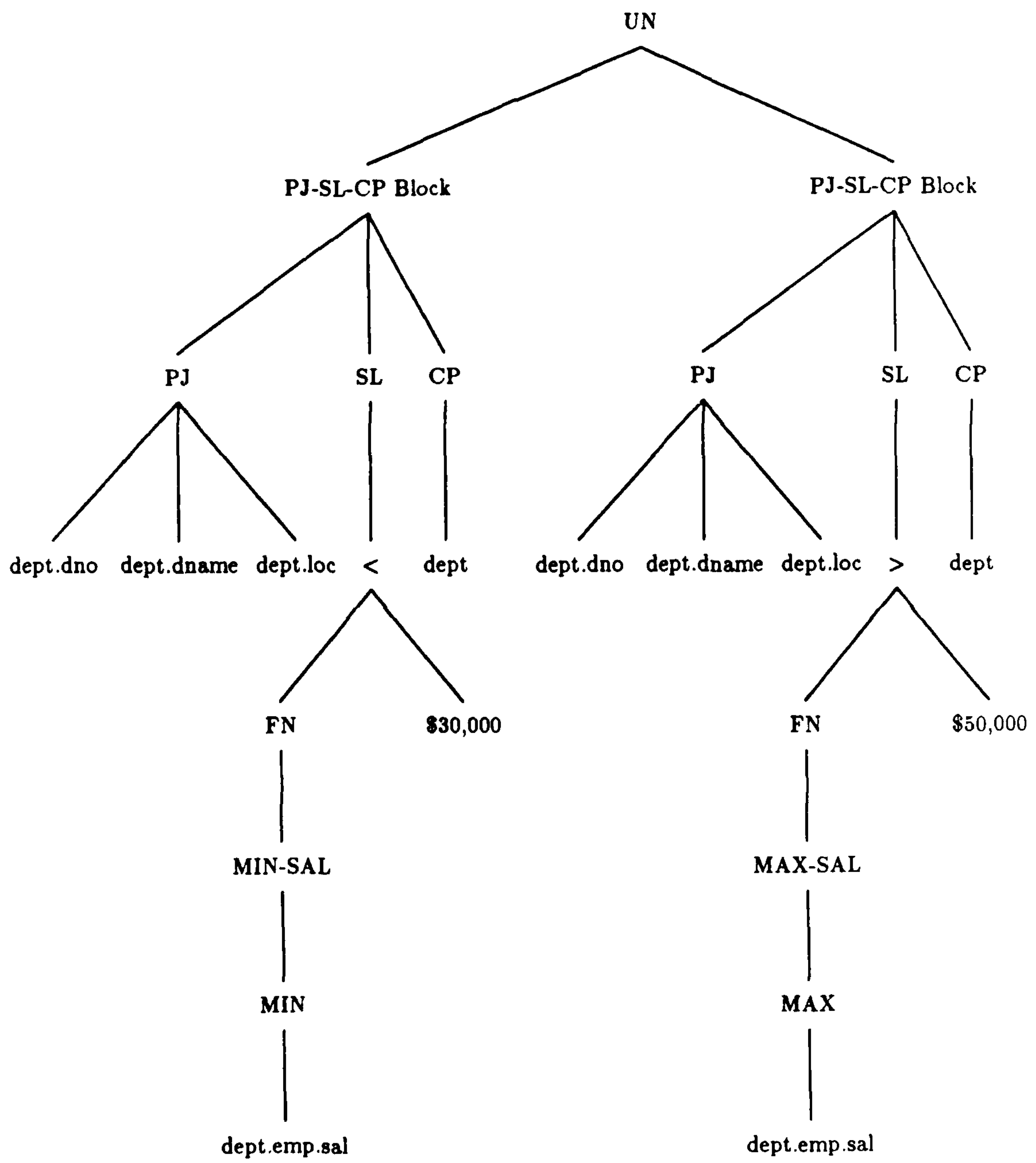


Figure 53. Example 7 Structure

DNO	DNAME	LOC
20	Finance	New York
30	Shipping	Dallas

Figure 54. Example 7 Output

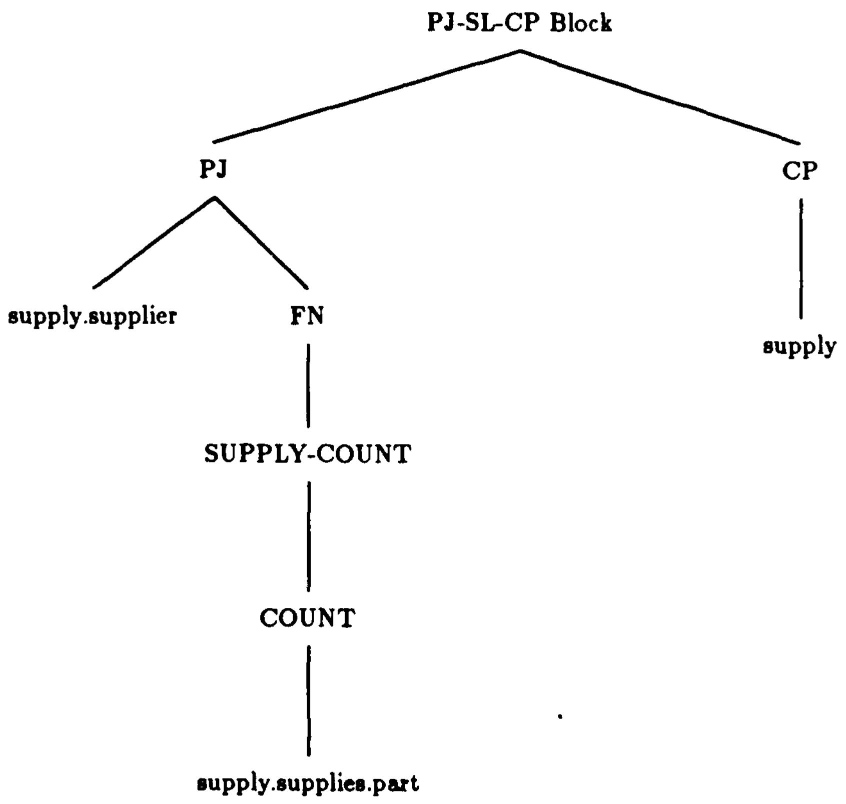


Figure 55. Example 8 Structure

SUPPLIER	SUPPLY-COUNT
National	15
Titan	15
Wilson	10

Figure 56. Example 8 Output

5.8 Performance Analysis

This section analyzes the performance of the algorithm in terms of order of the algorithm, also referred to as Big O [15]. The order of the algorithm is a measure of the time required for the algorithm to perform a query. The order of the algorithm is determined by analyzing each step of the algorithm to find how many times the step must be performed. The time required to perform a query will depend on the specific query and the size of the database. This section will analyze the algorithm by determining the order of each of the parts of the algorithm. The time required to perform a query will depend on the number of operators, operands, attributes and tuples involved in the query. In general, the number of tuples will be the major factor in determining performance because there are usually more tuples than operators, operands, and attributes. Another reason that tuples have a larger effect on performance is that the number of operators, operands, and attributes are constant for a given query and therefore occur as first order terms, whereas the number of tuples can occur as higher order terms. Thus, the primary factor in the analysis will be the number of tuples being processed.

The first step in the algorithm is construction of an intermediate data structure. The time required to construct the intermediate data structure for a query will depend on the number of nodes in the intermediate data structure. Because there is a node for each operator, each operand, and each descedant of an operand, the time required to build the intermediate data structure will depend on the total number of operators, operands and operand descendants in the query. Because the number of operators, operands and operand descendants is fixed for a given query, building the intermediate data structure has time O(1).

The time required to perform the Set_op_process is determined as follows. The time to perform step 1 of the Set_op_process will depend on the number of PJ-SL-CP blocks in the set operator expression. Because this number is a constant for a given query, step 1 will have time O(1). In order to perform step 2, each of the tuples in each PJ-SL-CP block must be compared to each of the tuples in the other PJ-SL-CP blocks. This tuple comparison consists of comparing each attribute in one tuple to each attribute in the other tuple. Therefore, the time required to perform step 2 will depend on the product of the number of tuples in each PJ-SL-CP block and the number of attributes in a tuple. Because step 1 is of constant order, the order of the Set_op_process is equal to the order for step 2 which is the product of the number of tuples in each PJ-SL-CP block and the number of attributes in a tuple.

The time required to perform the PJ-SL-CP_process is determined as follows. Steps 1 and 2 will occur once for each query and therefore have time O(1). The time required to perform step 2 is determined by using the SL_process order of analysis. The time required to perform step 3 is determined by using the PJ_process order of analysis. Steps 3 thru 5 will occur once for each combination of tuples in the Cartesian product. Therefore, steps 3 thru 5 will have an order equal to the product of the number of tuples in each relation times the sum of the SL_process and PJ_process. Because steps 3 thru 5 determine the performance of the PJ-SL-CP_process, the order of the PJ-SL-CP_process will be equal to the product of the number of tuples in each relation times the sum of the SL_process and PJ_process.

The time required to perform the SL_process is determined as follows. The time to perform step 1 depends on the number of leaf nodes in the SL branch of the intermediate data structure that represent field names. Because this number is a constant for a given query, step 1 has time O(1). The time to perform step 2 depends on the number of boolean and predicate operators in the SL branch of the intermediate data structure. Because this number is a constant for a given query step 2 has time O(1). The SL_process has time O(1), since each of the steps of the process has time O(1).

In the PJ_process, the time required to perform step 1 will depend on the number of field names in the PJ statement. Because the number of field names is a constant, step 1 has time O(1). The time required to perform step 2 is determined by using the order of analysis for the

New_name_process. The overall order of the PJ_process will be equal to the sum of the orders for step 1 and step 2.

The time to perform the New_name_process will depend on type of structure being given a name. Step 1 will always be performed and then one of steps 2 thru 7 will be performed depending on the new name structure. Step 1 will be performed only once and has time O(1). The performance for each of steps 2 thru 7 will depend on the order of the process performed in that step. For example, the order of step 2 will be equal to the order of the Print_trace process and the order of step 3 will be equal to the order of the Aggregate_process. Because step 1 has time O(1), the order of the New_name_process will be equal to the order of the step from 2 thru 7 that is performed.

The time required to perform the Aggregate_process is determined as follows. Step 1 will be performed once for each tuple in the aggregate relation. Therefore, the order of step 1 is equal to the number of tuples in the aggregate relation. Step 2 will be performed a maximum of one time and has a time O(1). The order of the Aggregate_process will be equal to the order of step 1, which is the number of tuples in the aggregate relation.

The time required to perform the NEST_process will depend on the number of tuples in the relation to be nested and the number of non-NEST attributes in a tuple. Step 1 and step 2 are performed only once and have a time O(1). Steps 3 thru 9 will be performed once for each possible pairing of two tuples in the relation. Step 7 requires that each non-NEST attribute be compared and has an order equal to the number of non-NEST attributes. Therefore, the order of steps 3 thru 9 will be the number of the non-NEST attributes times the square of the number of tuples in the relation to be nested. The order of the NEST_process will be equal to the order of steps 3 thru 9, which is the number of the non-NEST attributes times the square of the number of tuples in the relation to be nested.

The time required to perform the UNNEST_process will depend on the number of tuples in the relation and the number of nested entries in each nested tuple. Step 1 will be performed only once and has a time O(1). Steps 2, 3 and 5 will be performed once for each tuple in the relation. Step 4 will be performed once for each nested entry in the tuple. Therefore, the order of the UNNEST_process will be equal to the sum of the number of nested entries in all the tuples in the relation.

The time required to perform the Print_trace_process will depend on the number of fields that are printed. Steps 1 thru 4 of the Print_trace_process will print an attribute and all its descendents. Therefore, the order of the Print_trace_process will be equal to the sum of the attribute and all its descendents.

The performance analysis in this section has described the order of each part of the algorithm. This section has not provided an overall order for an entire query because the time required to perform a given query will depend on the specific query. However, based on the analysis in this section it is possible to determine which parts of the query will limit performance for queries in general. The parts of the algorithm that have time O(1) will have relatively little effect on the overall performance of the algorithm. The parts of the algorithm that have time O(1) are the SL_process, PJ_process and constructing the intermediate data structure. The parts of the algorithm that will have the largest effect on performance are parts which depend on the product of the number of tuples in one or more relations. These parts include the Set_op_process, PJ-SL-CP_process, NEST_process, and in some cases the New_name_process.

VI. Conclusion

6.1 Summary of Results

This thesis has presented an algorithm to convert nested relational algebra queries into GENSIS Trace Manager commands. The design of this algorithm is an important step in the development of a DBMS that supports nested relations. In many situations nested relations provide a more accurate representation of real world data than do 1NF relations. In addition, nested relations can provide a more efficient representation of data by reducing redundancy of data and simplifying the update of data.

The algorithm was validated by demonstrating that it correctly translates nested relational algebra queries into GENESIS Trace Manager commands. The first step in the validation was to divide nested relational algebra queries into different types of expressions. The next step was to provide example queries that included each type of expression. The final step in the validation was to show that the algorithm correctly translated each of the example queries into GENESIS Trace Manager commands.

The performance of the algorithm was evaluated by performing an order of analysis. The performance analysis included an analysis of each of the processes in the algorithm. The results of the analysis showed that the processes in the algorithm having the largest effect on performance are the Set_op_process, PJ-SL-CP_process, NEST_process, and in some cases the New_name_process.

6.2 Further Study

There are two primary areas that require further research. The first is to implement the algorithm and the second is to extend the algorithm. The algorithm needs to be extended because the GENESIS Trace Manager is limited to manipulating fields within records that have been read into buffers in primary memory. The GENESIS Trace Manager does not include facilities for accessing records in a database or producing formated output for queries. Because the GENESIS Trace

Manager does not include these facilities, the algorithm does not produce GENESIS commands to perform these functions. The algorithm needs to be extended to include facilities for:

1. Interfacing to lower level database functions so that records in a database can be accessed.
2. Generating formated output so the results of a nested relational algebra query can be presented to the user.

Appendix A. *GENESIS Trace Manger Terminating Conditions*

This appendix describes the normal termination conditions and error termination conditions for the GENESIS trace functions. The data in this appendix is from the GENESIS Record Manager User Manual [1, pages 22-24].

VALUE	EXPLANATION
BADLEVEL	attempt to attach trace variable at a level not higher than the present level of attachment
BOUNDS_VIOLATION	cannot go to the requested field or repeating group element, it is outside of the logical bounds
EMPTY_FIELD	cannot delete an element, repeating field is empty
LEAF_NODE	cannot go down, this is a leaf node
MISMATCH	cannot make a replacement, the source and target fields do not match
NON_RPG	cannot add or delete an element, this is not a repeating group
ROOT_NODE	cannot go up, right, or left, this is the root node
RW_ILLEGAL	cannot do an I/O operation because the trace is currently virtually positioned
UNKNOWN	cannot give requested information because the trace is currently unattached

Figure 57. Possible Fatal Error Values of Trace Functions [1, page 22]

VALUE	EXPLANATION
ABSENT_ELEMENT	cannot go to requested repeating group element, it is not present
NO_MORE_ROOM	cannot add an element, repeating field is full
OKAY	no error

Figure 58. Possible Normal Terminating Values of Trace Functions [1, page 22]

NORMAL TERMINATING CONDITION	EXPLANATION
OKAY	All functions

ABSENT_ELEMENT	down()
	down2()
	skip()
	left()
	right()

NO_MORE_ROOM	ad()

Figure 59. Trace Functions and Possible Normal Terminations [1, page 24]

FUNCTION	LIST OF FATAL ERROR CONDITIONS
Utility Functions	
attach_trace()	BADLEVEL
copy_trace()	
decode_str()	
encode_str()	
free_buf()	
free_trace()	
get_buf()	
get_trace()	
init_trace()	
print_trace()	
refresh_trace()	
reset_status()	
set_tr()	
Navigation Functions	
down()	LEAF_NODE BOUNDS_VIOLATION
down2()	LEAF_NODE BOUNDS_VIOLATION NON_RPG
field_left()	ROOT_NODE BOUNDS_VIOLATION
field_right()	ROOT_NODE BOUNDS_VIOLATION
left()	ROOT_NODE BOUNDS_VIOLATION NON_RPG
restore()	
right()	ROOT_NODE BOUNDS_VIOLATION NON_RPG
skip()	ROOT_NODE BOUNDS_VIOLATION
up()	ROOT_NODE
up2()	ROOT_NODE
Information Functions	
count()	UNKNOWN
ft()	
len()	UNKNOWN
loc()	UNKNOWN
rwok()	
status()	

Figure 60. Trace Functions and Possible Error Conditions [1, page 23]

FUNCTION	LIST OF FATAL ERROR CONDITIONS		
I/O Functions			
ad()	RW_ILLEGAL	NON_RPG	
dl()	RW_ILLEGAL	NON_RPG	EMPTY_FIELD
field_copy()	RW_ILLEGAL	MISMATCH	
mk()			
rd()			
rep()	RW_ILLEGAL	MISMATCH	

Figure 61. Trace Functions and Possible Error Conditions [1, page 24]

Bibliography

1. GENESIS Record Manager User Manual. Department of Computer Science, University of Texas at Austin.
2. D. S. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. *GENESIS: A Reconfigurable Database Management System.* Technical Report, Department of Computer Science, University of Texas at Austin, March 1986. TR-86-07.
3. S. Ceri and G. Gottlob. Translating SQL into Relational Algebra Optimization, Semantics, and Equivalence of SQL Queries. In *IEEE Transactions on Software Engineering*, pages 324–345, April 1985.
4. D. D. Chamberlin, A. M. Gilbert, and R. A. Yost. A History of System R and SQL/Data System. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 560–575, September 1981.
5. P. P. Chen. The Entity-Relationship Model–Toward a Unified View of Data. In *ACM Transactions on Database Systems*, pages 9–36, March 1976.
6. E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.
7. E. F. Codd. Further Normalization of the Data Base Relational Model. In *Data Base Systems:Courant Computer Science Symposia Series, Vol 6*, 1972.
8. E. F. Codd. Recent Investigations in Relational Data Base Systems. *Proceedings of the IFIP Congress*, 1017–1021, 1974.
9. E. F. Codd. Relational Database: A Practical Foundation for Productivity. *Communications of the ACM*, 25(2):109–117, February 1982.
10. C. J. Date. *An Introduction to Database Systems.* Volume II, Addison-Wesley, 1983.
11. C. J. Date. *An Introduction to Database Systems.* Volume I, Addison-Wesley, 1986.
12. R. Fagin. Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM Transactions on Database Systems*, 2(3):262–278, September 1977.
13. R. Fagin. Normal Forms and Relational Database Operations. *ACM SIGMOD International Conference on Management of Data*, 153–160, 1979.
14. Patrick C. Fisher and Stan J. Thomas. Operators for Non-First-Normal-Form Relations. In *Proceedings of the Computer Software and Applications Conference*, pages 464–475, Chicago, IL, November 1983.
15. Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures in Pascal.* Computer Science Press, 1984.
16. G. Jaeschke and H. Schek. Remarks on the Algebra of Nonfirst Normal Form Relations. *Proceedings of the ACM Symposium on Principles of Data Systems, Los Angeles*, 124–138, 1982.
17. G. Kappel, A. Tjoa, and R. Wagner. Form Flow Systems Based on NF2-Relations. In *Datembank-systems fur Buro, Technik und Wissenschaff*, pages 234–252, Informatik-Fachberichte Nr. 94, Springer-Verlag, Berlin, 1985.
18. W. Kent. A Simple Guide to Five Normal Forms in Relational Database Theory. *Communications of the ACM*, 26(2):120–125, February 1983.
19. Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* McGraw-Hill New York, New York, 1986.
20. David Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, Maryland, 1983.

21. A. Makinouchi. A Consideration on Normal Form of Not-necessarily-normalized Relations in the Relational Data Model. In *Proceedings of the Conference on Very Large Database Systems*, pages 447-453, Springer-Verlag, Berlin, 1977.

22. Z. Ozsoyoglu and Li-Yuan Yuan. A New Normal Form for Nested Relations. *ACM Transactions on Database Systems*, 12(1):251–260, 1987.

23. Z. Meral Ozsoyoglu and Gultekin Ozsoyoglu. A Query Language for Statistical Databases. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*, pages 171–187, Springer-Verlag, Berlin, 1985.

24. Srinivasan Ramakrishnan. *Design and Implementation of a Translator for SQL/NF with Role Joins.* Master's thesis, University of Texas at Austin, December 1986.

25. Mark Roth. *Theory of Non-First Normal Form Relational Databases.* PhD thesis, The University of Texas at Austin, Austin, TX, 1986.

26. H. A. Schmid and J. R. Swenson. On the Semantics of the Relational Model. *Proceedings of the ACM SIGMOD*, 1975.

27. Kenneth Paul Smith. *Design and Implementation of the GENESIS Record Manager.* Master's thesis, University of Texas at Austin, May 1985.

28. R. W. Taylor and R. L. Frank. CODASYL Data-base Management Systems. In *ACM Computing Surveys*, pages 67–103, March 1976.

29. D. C. Tsichritzis and F. H. Lochovosky. Hierarchical Data-base Management. In *ACM Computing Surveys*, pages 105–124, March 1976.

30. Jeffery D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville Maryland, 1982.

31. C. C. Yang. *Relational Databases.* Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

Vita

PII Redacted

Captain Alan F. Hartman was [redacted]. He graduated from Danville High School in Danville, Illinois in 1967 . He graduated from the University of Illinois at Urbana, Illinois in June, 1971 with a Bachelor of Science degree in Psychology. In 1971, he enlisted in the U.S. Army and served as a medical laboratory technician until 1974 when he received an honorable discharge from the Army. He studied undergraduate chemistry at the University of Illinois at Chicago, Illinois from 1974 to 1976 and graduate chemistry at the University of Illinois at Urbana, Illinois from 1977 to 1978 where he received a Master of Science degree in Chemistry. He entered Officer Training School in 1979 and received his commission in the USAF in July, 1979. After completing the Communication Systems Officer course in 1980, he served on a software design team at the Communications Computer Programming Center at Tinker AFB, Oklahoma. While working at Tinker AFB, he attended night school and received a Bachelor of Science degree in Computer Science in May, 1983. He served as a staff officer at HQ AFCC from August, 1984 thru May, 1986 at which time he entered the Air Force Institute of Technology.

[redacted]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS A190502
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/87D-13	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
See Box 19

12. PERSONAL AUTHOR(S)
Alan F. Hartman, M.S., Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM ________ TO ______	1987 December	96

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Data Bases, Algorithm
12	05		
05	02		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: DESIGN OF AN ALGORITHM TO TRANSLATE NESTED RELATIONAL ALGEBRA QUERIES TO GENESIS TRACE MANAGER COMMANDS

Thesis Chairman: Mark A. Roth, Captain, USAF
Assistant Professor of Computer Systems

Approved for public release: IAW AFR 190-1.
[illegible] 31 Dec 87
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AU)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Mark A. Roth, Captain, USAF	22b. TELEPHONE (Include Area Code) (513) 255-3576 22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

This thesis describes an algorithm to convert nested relational algebra queries into GENESIS Trace Manager commands. Nested relational algebra is an extension to traditional relational algebra to include multivalued (i.e. nested) attributes. The GENESIS Trace Manager is part of the GENESIS database management system being developed at the University of Texas at Austin, Texas. The GENESIS Trace Manager is used to manipulate fields in a record that has been read into a buffer in memory.

The algorithm consists of two phases. The first phase of the algorithm is the development of an intermediate data structure to represent the various constructs of the nested relational algebra query. The second phase of the algorithm is the conversion of the intermediate data structure into GENESIS Trace Manager commands. This phase consists of dividing the translation into a number of sub-tasks and providing an algorithm to perform each of these sub-tasks.

The GENESIS Trace Manager is limited to working with fields in a record located in a buffer in primary memory. It does not include facilities for reading records from a database into memory, writing records from memory to a database, or presenting the user with a formated output of the result of the query. Because the GENESIS Trace Manager does not include these facilities, the algorithm does not produce GENESIS commands to perform these functions.